Aufgabe 2: Rechenrätsel

DOKUMENTATION

Teilnahme-ID: 60291

Siemen Zielke

12. April 2022

Inhalt

Lösungsidee 1
Umsetzung 2
Schritt 1: Generieren der Zahlen 2
Schritt 2: Berechnen aller möglichen Rechenergebnisse 2
Schritt 3: Rechenergebnisse aussortieren 2
Schritt 4: Auswahl aus den übrigen Ergebnissen treffen 3
Beispiele 3
Quellcode 4

Lösungsidee

Ich teile die Generierung der Rätsel in 4 Schritte:

1. Pseudo-zufälliges Generieren der Zahlen vor dem Gleichheitszeichen.
2. Berechnen aller Rechenergebnisse, die durch Einsetzen unterschiedlicher Kombinationen von Rechenoperatoren zwischen den Zahlen erzeugt werden können.
3. Aussortieren der Rechenergebnisse, die
    a. eine oder mehrere nicht ganze Zahlen als Zwischenergebnis aufweisen
    b. negativ sind oder
    c. mehrfach vorkommen.
4. Auswahl eines der Rechenergebnisse und der dazugehörigen Rechnung nach dem Schema:
    a. Entferne alle Rechnungen, die weniger unterschiedliche Operatoren haben als die mit den meisten Operatoren. *Dies sortiert uninteressante Rätsel mit einfachen Lösungen aus.*
    b. Sortiere die übrigen Ergebnisse nach deren Distanz zum Median aller Ergebnisse. *Je näher ein Ergebnis am Median liegt, desto mehr Ergebnisse sind im näheren Umfeld und das Rätsel dadurch schwieriger. Ist ein Ergebnis sehr hoch, ist klar, dass viel Multiplikation verwendet werden muss. Wenn das Ergebnis eher in der Mitte liegt, gibt es viel mehr Möglichkeiten, Operatoren einzusetzen, die ein ähnlich hohes aber falschen Ergebnis erzielen. Man muss also mehr herumprobieren und das Rätsel ist schwerer.*
    c. Wähle ein zufälliges Ergebnis als Rätsel aus, aber so, dass die Ergebnisse weiter vorne in der Liste wahrscheinlicher ausgewählt werden als die weiter hinten.

Umsetzung

Hier folgt die Erläuterung meiner Implementierung der Lösungsidee in Python. Der gesamte Code von mir befindet sich in einem Modul. Bei anderen genannten Modulen handelt es sich um Module, die von Python bereitgestellt werden.

Schritt 1: Generieren der Zahlen

Ich benutze den Pseudozufallsgenerator random.randint(), um die vom Nutzer angegebene Menge an Zahlen von 1 bis 9 zu generieren und sie in einer Liste abzuspeichern.

Schritt 2: Berechnen aller möglichen Rechenergebnisse

Ich benutze die Funktion product() aus dem itertools-Modul, die einen Python-Generator erzeugt. Jede Iteration dieses Generators gibt eine Kombination aus den der Funktion übergebenen Werten als Tupel zurück. Die Länge des Tupels wird der Funktion übergeben.
Ich übergebe also der Funktion eine Liste mit den Strings "+", "-", "*" und "/" und die gewünschte Menge an Operatoren. Dann iteriere ich über den erhaltenen Generator.

Nun muss ich die Zahlen mit den jeweiligen Operatoren kombinieren und den Term dann ausrechnen. Wichtig dabei ist, dass ich Schritt 3.a der Lösungsidee, also das Überprüfen auf nicht ganze Zwischenergebnisse, schon hier implementieren muss, da im Nachhinein die Zwischenergebnisse nicht mehr verfügbar sind.

Ich setzte also als erstes die Operatoren mit den Zahlen zu einem String zusammen. Dieser String kann dann von der Python-Funktion *eval()* ausgerechnet werden. Das Problem dabei ist aber, dass ich dann keine Kontrolle mehr über die Zwischenergebnisse habe. Deswegen erstelle ich eine Kind-Klasse *NoFractionInt* von int. Ich überschreibe die Methode für den /-Operator so, dass eine Exception geworfen wird, wenn das Ergebnis keine ganze Zahl ist. Außerdem überschreibe ich den Typ der Rückgabewerte aller vier Operatoren +-*/ so, dass ein *NoFractionInt* und kein normaler int zurückgegeben wird. Damit die Klasse auch verwendet wird, umschließe ich jede Zahl im String mit dem Konstruktor *NoFractionInt(*Zahl*)*. Der String könnte also zum Beispiel "NoFractionInt(4)*NoFractionInt(4)-NoFractionInt(3)" sein.

Ich kann nun in einem try-Block die eval() Funktion mit dem generierten String ausführen. Im except-Block fange ich dann die Exceptions für nicht ganze Zwischenergebnisse ab und springe mit *continue* zur nächsten Operatorkombination. Danach überprüfe noch, ob das Ergebnis negativ ist (Schritt 3.b) und mache auch in dem Fall mit continue mit der nächsten Kombination weiter.

Wenn das Ergebnis die beiden „Tests“ überstanden hat, speichere ich es und die Operatorkombination in zwei separaten Listen. Dabei wandle ich das Ergebnis wieder zu einem int um.

Schritt 3: Rechenergebnisse aussortieren

Ich habe die Ergebnisse nach den ersten beiden Kriterien (3.a und 3.b) schon in Schritt 2 der Implementierung herausgefiltert. Es müssen also nur noch die doppelt vorkommenden Ergebnisse entfernt werden (3.c). Dafür mache ich zunächst eine Kopie von der Liste der Ergebnisse. Dann iteriere ich von hinten über die Ergebnisse. Für jedes Ergebnis zähle ich die Anzahl seines Vorkommens in der Kopie. Wenn das Ergebnis mehrmals vorkommt, entferne ich es aus der Originalliste. Dadurch, dass ich von hinten iteriere, verschieben sich die Indexe der noch nicht überprüften Ergebnisse nicht, wenn ich ein Ergebnis entferne.

Schritt 4: Auswahl aus den übrigen Ergebnissen treffen

Als Erstes entferne ich alle Operatorkombis, die weniger unterschiedliche Operatoren haben, als die mit den meisten. Dafür erstelle ich zunächst eine neue Liste, in der ich die Diversität (Anzahl der unterschiedlichen Operatoren) für jede Kombi speichere. Ich wandle dafür jeweils den Tupel mit der Operatorkombi in ein Set um. Da ein Set keine doppelten Elemente enthalten kann, ist die Länge dieses die Diversität der Kombi. Dann speichere ich den höchsten Wert dieser Liste in einer Variablen.

Danach fusioniere ich die drei Listen – Ergebnisse, Operatorkombis und Diversitäten – zu der Liste Rätsel mithilfe der Python-Funktion *zip()*. Über diese Liste iteriere ich dann wieder rückwärts. Ich vergleiche die Diversität des Rätsels mit der Höchsten. Ist sie kleiner entferne ich das Rätsel aus der Liste.

Nun müssen die übrig gebliebenen Rätsel sortiert werden. Dafür verwende ich die *sort()* Methode von Python-Listen. Dieser Methode kann man eine Funktion übergeben, die als Sortierkriterium dienen soll. Dieser Funktion werden am Anfang alle Elemente der Liste übergeben und diese muss dann einen Zahlenwert zurückgeben. Die Elemente werden dann in aufsteigender Reihenfolge ihrer Zahlenwerte sortiert.

Ich definiere also eine Funktion, die ein Rätsel als Argument nimmt und den Abstand (Betrag der Differenz) seines Rechenergebnisses zum Median aller Rechenergebnisse zurückgibt. Dafür berechne ich mithilfe der *median()* Funktion des *statistics* Moduls den Median aller Ergebnisse. Dabei beziehe ich auch die Ergebnisse, die durch geringe Diversität herausgefiltert wurden, mit ein, da diese auch den Kriterien für ein valides Rätsel entsprechen und somit auch Auskunft darüber geben, in welchem Zahlenbereich die meisten Ergebnisse liegen.

Jetzt kann ich die Liste mit der *sort()* Methode nach meinen Kriterien sortieren.

Als letztes muss noch ein Rätsel aus der Liste ausgesucht werden. Dafür benutze ich die *choices* Funktion vom *random* Modul. Dieser kann man eine Liste von Gewichten übergeben und diese wählt dann anhand der Gewichte ein zufälliges Element aus einer Liste. Für die Gewichte generiere ich einfach eine Liste mit den Zahlen n bis 1 (wenn n die Anzahl aller Rätsel ist). Das heißt jedes Rätsel ist immer $\frac{1}{\sum_{x=1}^{n} x}$ weniger wahrscheinlich als das davor.

Nachdem ich ein Rätsel ausgesucht habe, muss ich es nur noch einmal ohne Operatoren als Rätsel und einmal mit Operatoren als Lösung ausgeben.

Beispiele

Der Algorithmus braucht auf meinem Computer für 10 Operatoren etwa 2 Minuten. Da mit jedem neuen Operator die Menge der möglichen Operatorkombis vervierfacht wird, wird auch die Rechenzeit vervierfacht. Das heißt für 15 Operatoren werden schätzungsweise etwa 34 Stunden benötigt. Ich habe hier einfach ein paar Beispiele für Operatorenzahlen 1 bis 10 eingefügt:

```
8∘3=11
8+3=11

1∘7=7
1*7=7


3∘6∘4=22
3*6+4=22

7∘7∘4=10
7+7-4=10
```

```
5∘2∘7∘8∘2∘6∘9=332
5+2*7*8/2*6-9=332


1∘2∘7∘4∘8∘9∘2∘7=263
1+2*7-4+8*9/2*7=263

9∘9∘4∘7∘1∘3∘2∘6=303
9*9*4-7+1-3-2*6=303


3∘1∘6∘9∘2∘7∘7∘7∘2=1316
```

```
                                                3-1+6*9/2*7*7-7-2=1316

 1∘8∘7∘3=12                                     7∘5∘9∘5∘2∘9∘6∘4∘8=815
 1*8+7-3=12                                     7-5+9*5*2*9+6*4/8=815

 6∘6∘3∘5=34
 6*6+3-5=34                                     6∘7∘5∘2∘3∘1∘4∘1∘3∘8=341
                                                6*7*5/2*3-1+4-1+3*8=341

 9∘5∘6∘2∘7=41                                   7∘9∘9∘6∘8∘2∘7∘6∘3∘2=1515
 9*5+6/2-7=41                                   7-9+9*6*8/2*7+6-3+2=1515

 2∘3∘3∘7∘3=15
 2+3*3+7-3=15                                   9∘1∘1∘1∘4∘9∘5∘2∘4∘2∘9=1622
                                                9-1-1-1-4+9*5*2*4/2*9=1622

 9∘3∘4∘6∘1∘9=80                                 8∘8∘1∘2∘8∘7∘3∘6∘3∘3∘7=2037
 9/3*4*6-1+9=80                                 8+8-1+2*8*7*3*6-3/3+7=2037

 6∘7∘6∘5∘9∘3=89                                 2∘9∘3∘9∘2∘7∘1∘7∘5∘9∘6=1789
 6-7+6*5*9/3=89                                 2*9*3*9/2*7-1+7*5+9*6=1789

                                                8∘8∘8∘2∘1∘2∘8∘7∘3∘1∘4=590
 4∘2∘5∘6∘3∘6∘4=178                              8*8*8/2+1+2*8*7*3+1-4=590
 4/2*5*6*3-6+4=178
```

Quellcode

Der Quellcode befindet sich in *calculationpuzzle.py*. Beim Starten muss als erstes Argument die Anzahl der Operatoren übergeben werden. Ich füge hier die in der Umsetzung beschriebenen Teile des Codes ein. Der Rest dient nur zum Parsen der Argumente und der Ausgabe des Rätsels.

```
import argparse
from random import randint as _randint, choices as _choices
from itertools import product as _product
from statistics import median as _median

#int overloading hack for not allowing divison fraction results
class DivisionResultNotInteger(Exception): ...

class NoFractionInt(int):
    def __add__(self, __x: int) -> int:
        return NoFractionInt(super().__add__(__x))
    def __radd__(self, __x: int) -> int:
        return NoFractionInt(super().__radd__(__x))

    def __sub__(self, __x: int) -> int:
        return NoFractionInt(super().__sub__(__x))
    def __rsub__(self, __x: int) -> int:
        return NoFractionInt(super().__rsub__(__x))

    def __mul__(self, __x: int) -> int:
        return NoFractionInt(super().__mul__(__x))
    def __rmul__(self, __x: int) -> int:
        return NoFractionInt(super().__rmul__(__x))

    def __truediv__(self, __x: int) -> int:
        __r = super().__truediv__(__x)
        if __r.is_integer(): return NoFractionInt(__r)
        raise DivisionResultNotInteger()
    def __rtruediv__(self, __x: int) -> int:
        __r = super().__rtruediv__(__x)
        if __r.is_integer(): return NoFractionInt(__r)
        raise DivisionResultNotInteger()

#parse arguments […]

#generate numbers
numbers = [_randint(1, 9) for i in range(digits)]
```

```
#calculate results for substituting every possible operation combination between "numbers"
results = []
operation_lists = []
for operations in _product(["+", "-", "*", "/"], repeat=digits - 1):
    operations = list(operations)
    expression = "".join([f"NoFractionInt({x}){y}" for x, y in zip(numbers, operations)]) +
f"NoFractionInt({numbers[-1]})"
    try: result = eval(expression)
    except DivisionResultNotInteger: continue
    if result <= 0: continue
    results.append(int(result))
    operation_lists.append(operations)

#remove duplicated solutions
results_with_duplicates = results.copy()
for i, result in reversed(list(enumerate(results))):
    if results_with_duplicates.count(result) > 1:
        results.pop(i)
        operation_lists.pop(i)
del results_with_duplicates

#remove too boring solutions
diversitys = [len(set(x)) for x in operation_lists]
puzzles = list(zip(results, operation_lists, diversitys))
best_diversity = max(diversitys)
for i, solution in reversed(list(enumerate(puzzles))):
    if solution[2] < best_diversity:
        puzzles.pop(i)
del best_diversity

#sort list (after their distance of their result to the median of all results)
median = _median(results)
def solutionSorter(puzzle):
    return abs(median - puzzle[0])
puzzles.sort(key=solutionSorter)
del median, solutionSorter

#select solution
weights = list(reversed(list(range(1, 1+len(puzzles)))))
output_puzzle = _choices(puzzles, weights=weights)[0]

#output puzzle […]
```