

Aufgabe 1: Wörter aufräumen

Team-ID: 00383

Team: Theodor Tesla

Bearbeiter dieser Aufgabe:
Theodor Tesla

23. November 2020

Inhaltsverzeichnis

Lösungsidee.....	1
Umsetzung.....	2
SearchedWord.....	2
recursiveSolution(searched, given, cat).....	2
wouldFit(hidden, openned).....	3
fixDuplicateWords(searched, given, cat).....	3
Anwendung.....	4
Beispiele.....	4
Beispielhafte Durchführung des Programms.....	4
Vorgegebene Beispiele (Rätsel der BwInf Seite).....	5
Quellcode.....	6
recursiveSolution(searched, given, cat).....	6
fixDuplicateWords(searched, given, cat).....	6

Lösungsidee

Meine Lösungsidee besteht darin, dass ich herausfinde, welche Wörter eindeutig zuweisbar sind, diese dann zuzuweisen und dies immer weiter zu wiederholen.

Dieser Algorithmus ist in zwei Schritte aufgeteilt. Zuerst wird überprüft, ob es Wörter gibt, die schon eindeutig sind, das heißt bei denen sich sagen lässt, dass sie aufgrund der Länge und der vorgegebenen Buchstaben der „verschlüsselten“ Wörter nur zu einem Wort passen.

Darauf folgend wird geprüft, ob das gleiche Wort mehrmals in der Lösung vorkommt und ob die Möglichkeiten, wo dieses Wort hinpassen würde gleich der Anzahl der Vorkommen des Wortes ist. Das Problem, welches ohne diesen Schritt aufkäme, wäre, dass der Algorithmus denkt, dass es für eines der mehrfach vorkommenden Wörter mehrere Möglichkeiten gibt. Da diese verschiedenen möglichen Positionen aber für das gleiche Wort sind, sollte es eigentlich eindeutig zuweisbar sein.

Umsetzung

Die genutzte Sprache war Python. Um die Prüfung der Eindeutigkeit zu vereinfachen habe ich zuerst die gegebenen Wörter in ein Dictionary umgewandelt. Hierbei war der Wert jeweils ein List-Objekt mit den Indices der Wörter, welche die Länge haben, die der Schlüssel darstellt. Die gesuchten Wörter werden in einem Datentypen namens *SearchedWord* gespeichert.

Der Algorithmus wird durch eine for-each-Schleife dargestellt, welche durch alle Schlüssel-Werte-Paare des Dictionarys für die zu suchenden Wörter iteriert. Dort wird bei jeder Iteration die Methode *recursiveSolution(searched, given, cat)* aufgerufen.

SearchedWord

Diese Klasse besitzt folgende Attribute:

word	Speichert das verdeckte Wort mit den Unterstrichen anstatt der Buchstaben
punctuation	Speichert das Satzzeichen am Ende des Wortes; Falls keines vorhanden → Leerer String
completedWord	Erstmals leer; Speichert das zugehörige Wort, sobald es zugeordnet wird
givenChars	Dictionary{int : char} Speichert den Index des vorgegebenen Buchstabens
origIndex	Speichert den Index des Wortes in dem fertigen Satz

Der Konstruktor besitzt die Parameter *newWord* und *index*; jeweils als Wert für die Attribute *word* und *index*.

Die Attribute sind alle public, da ich in anderen Sprachen (bspw. C++) ein **struct** genutzt hätte, dass nur zur Speicherung/Sammlung von Werten da ist und dessen Attribute deshalb auch public sind.

recursiveSolution(searched, given, cat)

Parameter:

searched	Dictionary{ int : List[SearchedWord] } <i>Schlüssel:</i> Länge der Wörter <i>Wert:</i> Liste mit gesuchten Wörtern; alle gleich lang
given	Dictionary{ int : List[SearchedWord] } <i>Schlüssel:</i> Länge der Wörter <i>Wert:</i> Liste mit gegebenen Wörtern; Alle gleich lang
cat	Int <i>Wert:</i> Schlüssel für searched und given

Die Prüfung läuft so ab, dass durch eine geschachtelte for-Schleife geprüft wird, ob ein gegebenes Wort zu einem gesuchten Wort passen würde. Hierbei stellt die Zählvariable der äußeren Schleife

den Index der gegebenen Wörter dar und die innere Zählvariable die gesuchten Wörter. Dies wird durch die Methode *wouldFit(hidden, opened)* erreicht. Wenn diese Methode *True* zurückgibt, wird eine Zählvariable erhöht, so dass man, nachdem durch die innere Schleife durchgelaufen wurde, weiß, an wie vielen möglichen Positionen dieses Wort stehen kann. Sollte dies 1 sein, weiß man, dass es nur eine Möglichkeit gibt, so dass diese dann zugewiesen werden kann.

Nach der äußeren Schleife wird geprüft, ob alle gegebenen Wörter einer Größe einem gesuchten Wort zugeordnet werden konnten. Wenn dies der Fall ist, wird die Schleife einfach abgebrochen und der Durchlauf beginnt von vorne mit der nächsten Größe an Wörtern, ansonsten wird die Methode *fixDuplicateWords(searched, given, cat)* und danach wieder *recursiveSolution(searched, given, cat)* aufgerufen.

wouldFit(hidden, opened)

hidden	SearchedWord Objekt, das ein gesuchtes Wort speichert, welches überprüft werden soll
given	String Objekt, das ein gegebenes Wort speichert, welches überprüft werden soll

Zuerst wird geprüft, ob die Länge der Wörter gleich ist. Dies sollte aufgrund der Sortierung nach Größe der Fall sein, ist aber etwas bei der Sortierung falsch gelaufen, zerstört es nicht sofort das Programm. Im weiteren Verlauf wird überprüft, ob es einen vorgegebenen Buchstaben gibt und falls nicht, wird der Wert *True* zurückgegeben. Ansonsten iteriert der Algorithmus durch die Schlüssel des Dictionarys, welches die gegebenen Buchstaben speichert und überprüft, ob diese auch so in dem gegebenen Wort an der Stelle vorkommen und gibt den dementsprechenden Wert zurück.

fixDuplicateWords(searched, given, cat)

searched	Dictionary{ int : List[SearchedWord] } <i>Schlüssel:</i> Länge der Wörter <i>Wert:</i> Liste mit gesuchten Wörtern; alle gleich lang
given	Dictionary{ int : List[SearchedWord] } <i>Schlüssel:</i> Länge der Wörter <i>Wert:</i> Liste mit gegebenen Wörtern; alle gleich lang
cat	Int <i>Wert:</i> Schlüssel für searched und given

Durch mehrere Schleifen wird aus **given** eine zwei-dimensionale Liste gebildet, die als zweite Dimension eine Liste mit Indices hat, die die Positionen von gleichen Wörtern in **given** darstellen; Die gleichen Wörter sind in der ersten Dimension gespeichert. Diese Liste besteht nur aus Listen mit mindestens zwei gleichen Wörtern, da alles andere keinen Sinn ergäbe.

Dann wird diese Liste so ähnlich, wie in der Methode *recursiveSolution(searched, given, cat)* gelöst. Der Unterschied ist bloß, dass das Einfügen der gegebenen Wörter in die gesuchten dann stattfindet, wenn die Anzahl der passenden Wörter gleich der Anzahl der gleichen Wörter ist, so dass es klar ist, dass diese Menge an gleichen Wörtern zu genau diesen gesuchten Wörtern gehört. Sollten es mehr Möglichkeiten sein, wird dies durch den nächsten Aufruf von *recursiveSolution(searched, given, cat)* gelöst, da das Rätsel ansonsten nicht eindeutig wäre, da mehrere Wörter zu mehreren gesuchten Wörtern gehören können.

Anwendung

Auf dem Betriebssystem MacOS X (10.15.7) funktioniert das Testen des Programmes wie folgt:

- Man geht davon aus, dass der Arbeitspfad im Terminal *Users/bwinf* ist
- Außerdem befindet sich die *main.py* Datei im Ordner *Users/bwinf/2020/Aufgabe1/TheodorTesla/Aufgabe1/src* und die Eingabedatei im Ordner *Users/bwinf/2020/Aufgabe1/TheodorTesla/Aufgabe1/examples*
- Im Terminal dann eingeben:
- `cd 2020/Aufgabe/TheodorTesla/Aufgabe1`
`python3 src/main.py raetsel0.txt`
- Dies gibt dann die Lösung der Datei *raetsel0.txt* aus
- Um eine andere Eingabe-Datei zu verwenden, muss man das *raetsel0.txt* mit einer *.txt* Datei austauschen, die sich im *examples* Ordner befindet

Beispiele

Beispielhafte Durchführung des Programms

beispiel0.txt:

`__s __t e__ _____l, d__ __t __!`

`toll das ist Beispiel ist ein das`

1. Kategorisierung:
 1. { 3: ['das', 'ist', 'ein', 'ist', 'das'], 4: ['toll'], 8: ['Beispiel'] }
 2. { 3: [__s, __t, e__, d__, ____], 4: [____t], 8: [_____] }
2. Erste Eindeutigkeit finden:
 1. toll => __t; Beispiel => _____l

3. Doppelte finden und eindeutige zuordnen:

1. Zweidimensionale Liste, die doppelte Wörter speichert: [[0, 4], [1, 2]] (*Indices aus given*)

1. ist => __t; ist => ____

2. das => __s, das => d__; ein => e__

==> **Alles zugeordnet: „das ist ein Beispiel, das toll ist!“**

Text aus der Konsole (MacOS X):

```
Aufgabe1 % python3 src/main.py beispiel0.txt
```

```
das ist ein Beispiel, das toll ist!
```

Vorgegebene Beispiele (Rätsel der Bwlnf Seite)

```
Aufgabe1 % python3 src/main.py raetsel0.txt
```

```
oh je, was für eine arbeit!
```

```
Aufgabe1 % python3 src/main.py raetsel1.txt
```

```
Am Anfang wurde das Universum erschaffen. Das machte viele  
Leute sehr wütend und wurde allenthalben als Schritt in die  
falsche Richtung angesehen.
```

```
Aufgabe1 % python3 src/main.py raetsel2.txt
```

```
Als Gregor Samsa eines Morgens aus unruhigen Träumen  
erwachte, fand er sich in seinem Bett zu einem ungeheueren  
Ungeziefer verwandelt.
```

```
Aufgabe1 % python3 src/main.py raetsel3.txt
```

```
Informatik ist die Wissenschaft von der systematischen  
Darstellung, Speicherung, Verarbeitung und Übertragung von  
Informationen, besonders der automatischen Verarbeitung mit  
Digitalrechnern.
```

```
Aufgabe1 % python3 src/main.py raetsel4.txt
```

```
Opa Jürgen blättert in einer Zeitschrift aus der Apotheke und  
findet ein Rätsel. Es ist eine Liste von Wörtern gegeben, die  
in die richtige Reihenfolge gebracht werden sollen, so dass  
sie eine lustige Geschichte ergeben. Leerzeichen und  
Satzzeichen sowie einige Buchstaben sind schon vorgegeben.
```

Quellcode

recursiveSolution(searched, given, cat)

```
def recursiveSolution(searched, given, cat):
    for iterate in range(len(given[cat]) - 1, -1, -1): # Iteration through words in a category
        counter = 0
        hiddenIndex = int
        for jIter in range(len(searched[cat])): # Check w/ every hidden word if it would match
            if wouldFit(searched[cat][jIter], given[cat][iterate]) and searched[cat]
[jIter].completedWord == '':
                counter += 1
                hiddenIndex = jIter
        if counter == 1:
            searched[cat][hiddenIndex].completedWord = given[cat][iterate]
            del given[cat][iterate]
            continue
    if len(given[cat]) != 0: # See notes for logical reference
        fixDuplicateWords(searched, given, cat)
        recursiveSolution(searched, given, cat)
    return
```

fixDuplicateWords(searched, given, cat)

```
def fixDuplicateWords(searched, given, cat):
    duplicateMem = []
    alreadyUsed = []
    for iterate in range(len(given[cat])): # Save occurrences of duplicate word in array
        duplicateMem.append([]) # First Index: Position of word in given[cat]
        duplicateMem[iterate].append(iterate) # Second Index: Positions of duplicates
        for jIter in range(iterate+1, len(given[cat])): # Duplicates aren't written twice...
            if given[cat][iterate] == given[cat][jIter] and jIter not in alreadyUsed:
                duplicateMem[iterate].append(jIter) # ... (i.e. [[0,2], [1], [0, 2]] would not
occur
                alreadyUsed.append(jIter) # It would be: [[0,2], [1], [2]]

    for subArr in range(len(duplicateMem)-1, -1, -1): # deleting everything that doesn't have a
duplicate
        if len(duplicateMem[subArr]) == 1:
            del duplicateMem[subArr]
```

```
for iterate in range(len(duplicateMem) - 1, -1, -1): # Iteration through duplicate words
    counter = 0
    hiddenIndex = []
    for jIter in range(len(searched[cat])): # Check w/ every hidden word if it would match
        if wouldFit(searched[cat][jIter], given[cat][duplicateMem[iterate][0]]) and
searched[cat][jIter].completedWord == '':
            counter += 1
            hiddenIndex.append(jIter)
    if counter == len(duplicateMem[iterate]): # Check if there are as many possibilities as
duplicate words
        for duplicateWord in range(len(duplicateMem[iterate])-1, -1, -1): # For every word,
mark as finished...
            # searched[cat][duplicateMem[iterate][duplicateWord]].completedWord = given[cat]
[duplicateMem[iterate][duplicateWord]]
            searched[cat][hiddenIndex[duplicateWord]].completedWord = given[cat]
[duplicateMem[iterate][duplicateWord]]
            del given[cat][duplicateMem[iterate][duplicateWord]] # ...(mark as finished) &
delete

        for jIter in range(len(duplicateMem)): # Loweres every index that is higher than
the index that deletes
            for kIter in range(len(duplicateMem[jIter])): # Else it would point to wrong
element because elements shift on eto left
                if duplicateMem[jIter][kIter] > duplicateMem[iterate][duplicateWord]: # But
indexes would't

                    duplicateMem[jIter][kIter] -= 1
```