Aufgabe 2: Schwierigkeiten

Programmnutzung

So verwendet man das Programm im Terminal:

```
$ python3 ./schwierigkeiten.py NAME_DER_EINGABEDATEI
```

Beispiel:

```
$ python3 ./schwierigkeiten.py ./schwierigkeiten0.txt
```

Grundgedanken und Programmablauf

Um wie in Aufgabe 2 gefragt die Aufgaben von einfach nach schwierig zu sortieren, haben wir uns für ein Ranking-System entschieden. Visualisieren kann man sich dieses, indem man sich vorstellt, die Aufgaben würden sich jeweils miteinander "duellieren". Die Prozedur ist wie folgt:

1. zuerst die Datei einlesen und parsen:
    1. nach Zeilenumbrüchen teilen, für jede Zeile:
        1. ist dies die erste Zeile? Wenn ja: Integer parsen und in Variablen laden
        2. ist dies eine Zeile ohne Vergleichsangaben - also widerum die letzte? Wenn ja: die in dieser Zeile angegebenen, zu sortierenden Werte speichern und die Schleife verlassen
        3. Nichts von all dem? Die Zeile an jedem "<" spalten und in eine Liste laden, dabei die gefundenen Aufgabennamen (wenn noch nicht zuvor geschehen) in ein Set laden, um den Überblick über alle existenten Aufgaben zu behalten.
    2. die gefundenen Daten an die `main`-Methode zurückgeben
2. dann ein Dictionary mit allen möglichen Scores vorbereiten, die bei 100 beginnen (der Wert muss grundsätzlich nicht genau 100 sein, wurde hier aber zur besseren Veranschaulichung gewählt.)
3. nun beginnen, das Scoring-System zu durchlaufen. Für jede Klausur:
    1. die einzelnen Aufgaben in der Klausur von (in der Datei) links nach rechts durchlaufen.
        1. für jede der Klausurzeilen wissen wir sicher: die Aufgabe rechts von derjenigen, bei der wir gerade sind, ist kniffliger. Also geben wir das dementsprechend an; wir vergeben ihr den Punktestand der Aufgabe, bei der wir gerade sind, und addieren 1 dazu, um für einen Abstand zu sorgen und nicht am Ende überall dieselben Scores zu haben.
4. die Ergebnisse nach den Scores sortieren, um diese dann letztendlich ausgeben zu können:
    1. führt normales Insertion Sort durch; extra Trickmagie ist nötig, um ein dict sortieren zu können:
        1. eine Liste erstellen, die die Keys des Dictionaries hält

    2. diese Liste mit den Keys mithilfe von Insertion Sort und Zurückgreifen auf das Original-Dict sortieren
    3. daraus wieder ein Dictionary erstellen - diesmal mithilfe von OrderedDict.
  2. sortiertes Ergebnis zurückgeben

5. das Ausgeben wird wie folgt bewerkstelligt - für jedes Element des Ergebnisses:

    1. ist die Aufgabe in den unter 1. ermittelten zu sortierenden Aufgaben? Wenn nein: zum nächsten
    2. wenn ja: Ausgeben an stdout, zur finalen Reihenfolgen-Liste hinzufügen
    3. Reihenfolgen-Liste, getrennt von "<"-Zeichen zurückgeben.

Anzumerken ist, dass wir für die Sortiermethode auch einfach `sorted()` hätten nutzen können. Dies hätte - zumindest in CPython - TimSort als Sortieralgorithmus verwendet, der dank C-Bindings und bei längeren Listen auch durch seinen effizienteren Algorithmus vermutlich um Längen schneller mit den Sortieren fertig gewesen wäre.

Da wir aber so viel wie möglich selbst implementieren sollten, haben wir den Sortieralgorithmus jetzt mal "mitgeliefert"! Um TimSort zu verwenden, müsste die Schleife schlichtweg so abgeändert werden:

```
...
for k, v in dict(sorted(scores.items(), key=lambda item: item[1])).items():
    ...
```

Wichtigste Teile des Quelltextes

Der mitunter wichtigste Teil ist das Ranking der einzelnen Aufgaben. Dieser sieht wie folgt aus:

```
def determine_easiest(lines: "list[list[str]]", scores: "dict[str, int]") -
> None:
    """
    Ermittelt die einfachsten Aufgaben durch ein Scoring-System.
    Aufgaben, die schwerer sind als andere, haben einen höheren Score.
    """

    for line in lines:
        # wir gehen jede Zeile durch
        line_len: int = len(line)
        for idx in range(line_len):
            if idx >= line_len - 1:
                # wenn wir am Ende der Zeile sind, können wir abbrechen
                break

            # wir holen uns die aktuelle und die nächste Aufgabe:
            cur_task: str = line[idx]
            next_task: str = line[idx + 1]
            # wenn die nächste Aufgabe in der Zeile schwerer ist, als die
aktuelle,
```

```
            # dann erhöhen wir den Score von der schwereren Aufgabe
            scores[next_task] = scores[cur_task] + 1
```

Wie wir sehen, ist eine Liste an Zeilen mit widerum Listen an Aufgaben (so wie im Eingabedokument) mitsamt des Scoring-Dictionaries (siehe oben) gegeben. Der Algorithmus ordnet dann die Elemente passend ein.

Ausgaben des Programms

Folgend die Ausgaben des Programms bei allen Beispielen:

```
user@testpc:~/bwinf$ python3 schwierigkeiten.py schwierigkeiten0.txt
B: 100
E: 102
D: 103
F: 105
C: 106
B < E < D < F < C
user@testpc:~/bwinf$ python3 schwierigkeiten.py schwierigkeiten1.txt
A: 100
C: 102
G: 102
F: 103
D: 104
A < C < G < F < D
user@testpc:~/bwinf$ python3 schwierigkeiten.py schwierigkeiten2.txt
B: 101
G: 102
A: 102
D: 102
E: 103
F: 104
B < G < A < D < E < F
user@testpc:~/bwinf$ python3 schwierigkeiten.py schwierigkeiten3.txt
I: 101
N: 101
L: 102
M: 102
J: 102
C: 102
K: 103
B: 103
H: 103
D: 104
A: 105
E: 106
F: 107
G: 108
I < N < L < M < J < C < K < B < H < D < A < E < F < G
user@testpc:~/bwinf$ python3 schwierigkeiten.py schwierigkeiten4.txt
B: 101
```

```
I: 103
F: 106
N: 106
W: 110
B < I < F < N < W
user@testpc:~/bwinf$ python3 schwierigkeiten.py schwierigkeiten5.txt
Z: 100
R: 100
H: 100
Q: 101
L: 101
C: 102
K: 102
X: 103
S: 103
J: 103
E: 104
O: 104
U: 105
M: 105
N: 105
T: 106
G: 106
P: 107
F: 107
B: 108
V: 108
D: 109
A: 109
W: 109
I: 110
Y: 111
Z < R < H < Q < L < C < K < X < S < J < E < O < U < M < N < T < G < P < F <
B < V < D < A < W < I < Y
```