One-Dimensional Scrabble

Rules of the game

Scrabble is a well known board game that people of all ages enjoy playing around the world both competitively and socially. Unfortunately, the full two-player game itself with all that branching string matching in two dimensions seems a bit too complicated. However, by reducing this game **to a single dimension and turning it from a two player game to a one player puzzle** so that this one player gets to make all the moves enables us to tackle this challenge into submission.

One horizontal row of the Scrabble board is given as a Python string guaranteed to consist of characters `'-'` used to denote blank spaces (minus signs are far easier to read and copy from console outputs than whitespace), and English lowercase letters `a-z` so that the row does not yet contain two letters anywhere without at least one blank character between them. One example of such a row of length 30 might be `'--o----v---c---i--b-l--i-n-u-y-'`.

Unlike in the actual Scrabble, you have an unlimited supply of every letter at your disposal. Your task is to replace any of the blanks in the row with letters of your choice to fill that row with words that maximize the overall score for the row. Unless you are lucky enough to find a word that fits the entire row, you still need to leave in some blanks between the words that you construct. Just like in Scrabble, **every maximal block of consecutive letter characters longer than one character must be some legal word** that can be found in our wordlist.

One possible way to fill in the previous row might be `'-convolvulic-trilobal-signeury-'`. If our dictionary is limited to contain only words of maximum length of seven letters, one good legal way to fill in this row might be `'azoxy-everich-bisabol-bignou-y-'`... still coming up with all those "five-dollar words" that your instructor cannot remember ever seeing anywhere.

The returned string is scored by adding up the individual scores for the words inside it. The score of each word is the sum of the Scrabble values of its letters. However, to make this problem more interesting so that you won't just scan for suitable positions to fill with short words of plenty of high-valued letters (such as `'zizz'` or `'xyzzy'`), each word is additionally scored for its length, so that a word with length `k` scores `k*k` points, making longer words significantly more valuable than short words, even if made of common letters worth a point or two each. A word of thirty characters would, by its sheer length, ring in a jackpot of 900 points! However, of course you try to maximize your total score by favouring words made of higher-valued letters if you get to choose between words of equal length.

Tester script

The GitHub repository ikokkari/ScrabbleRow contains the automated tester script scrabblerun.py that you can use to automatically test and evaluate your submission, along with the wordlist file words_sorted.txt. The row patterns given to your function to fill are generated pseudorandomly using the `seed` value defined in the beginning of the tester script. The game is played for the given number of `rounds` for patterns of length `patlen`, with the overall wordlist restricted to use the subset of words in `words_sorted.txt` whose length is between `minlen` and `maxlen`, inclusive.

An example run with the instructor's private model solution produced the following output. The score for each returned result is displayed in parentheses after the result.

```
Scrabblerun tester by Ilkka Kokkarinen, August 22, 2020.
Settings seed=123456, patlen=40, rounds=10.

Pattern: -a---t--a--l---t-t-a--a-k-d-s-----s-b--n
Result : labiotenaculum-tithal-akkadist-firstborn (437)

Pattern: -m-h-e--t----p---y-b-e--m--m-m-p-c-e-a-i
Result : amphierotism-pozzy-beedom-mmmm-puchera-i (356)

Pattern: --m-t-e-a--e-o---p-p-n-u-i--r-f--o-i--p-
Result : comitje-acneform-pipunculid-r-f-podiceps (342)

Pattern: -i--e-g--r-c-d-t-o--s-i--r-s-p---i-s--t-
Result : jisheng-trucidation-seizures-p-shipsmith (383)

Pattern: -r-n-t--p-s-o--c--e-e-i-i--i-o-l-o-e-s--
Result : prankt-episcopicide-exilic-i-onlookers-- (358)

Pattern: -e-u-r--o-o-a-e-l-s----a-d----b-i-l-a-e-
Result : zeburro-ozonate-l-sluggardize-bailliage- (369)

Pattern: -c--i---n-h-s---a-r----n-s-i-q-n--o---l-
Result : scutibranch-s-clairvoyants-i-q-nemophily (405)

Pattern: -i-i-c-o--r-e-c-l-e-a-n-o--g----b-p--o-c
Result : fiji-chozar-euchlaena-neologize-b-p-zoic (312)

Pattern: ----o-r-e--r-o-e-s-m--i-t--i-r-u-u-e-e-t
Result : coproprietor-oversum-historier-ukulele-t (376)

Pattern: -s-d-h-i-i-l-c--w-t-r-s--e-s--v-c-e-r--a
Result : -sidth-idiolect-waterishness-evicke-raja (349)
3687 Kokkarinen, Ilkka 123456789
```

As you can see in the above run, your function does not have to fill in the entire row. If you have no good word available for positions that contain a problematic letter, you are allowed to leave such letters as they were, even if those letters aren't considered one-letter words by themselves. You can also leave in blank spaces that consist of more than one blank space, even though this does not show in the above run.

The list line printed by the tester displays the total score followed by the information about the author of the solver. The higher the score, the better.

For the tester script to be able to find and interact with your code, all your code must be written in a file named `scrabblerow.py`. This file **must contain the following three functions exactly as specified**, plus whatever other helper functions of your own that you choose to write:

```
def author():
```

Returns the name of the author of this script, as string of the form `"Lastname, Firstname"`.

```
def student_id():
```

Returns the student ID of the author of this script as a string.

```
def fill_words(pattern, words, scoring_f, minlen, maxlen):
```

The function receives as parameters the `pattern` as a string and `words` as a list of words whose length is between `minlen` and `maxlen`, inclusive, and a scoring function `scoring_f` that will tell you how many points its argument word will score. From this information, the function returns a new string for the completed row.

The returned result string must be the same length as `pattern`, and in every position where `pattern` has a non-blank letter, the result must contain that same letter. Furthermore, each complete word inside the string must be a legal word from `words_sorted.txt` whose length is between `minlen` and `maxlen`, inclusive. Result strings that do not obey these constraints receive zero points. (The rest of the evaluation will still continue, though.)

To get started with this project, you should first implement the following non-optimal **greedy algorithm**, even if you plan to solve this problem in some other way. Implementing this algorithm and seeing it work correctly means that your subroutines that find all words that potentially fit into the given position and try them out work correctly. Having these subroutines available and known to work will then simplify your later journey into different ways to fill in the row.

The greedy algorithm for this problem starts from the beginning of the row. Loop through all the words that would fit into the pattern starting from that position, and choose the

highest-scoring of all such fitting words. Stamp that word into the pattern and continue filling the rest of the pattern from the next available position. Repeat until you have filled in the pattern so that there is no more room in the end to fit in any additional words.

This greedy approach produces a reasonable result for most instances reasonably quickly, but will not be optimal. Once you realize why this is the case, and what types of situations the greedy algorithm would grossly mishandle compared to the optimal way of filling the given pattern with letters, you are on your way of thinking up better ways to handle the job.

Grading the submissions

Each submission is evaluated with `rounds=20` and a secret `seed` value chosen by the instructor. The functions should be fast enough so this test completes its run in less than five minutes. The same `seed` is used for evaluating all submissions under identical conditions, so that the submissions can be ranked fairly based on their overall scores.

The submission with the highest score will receive the highest possible project mark of 20 points. Reading these submissions down the line along the decreasing score, the instructor finds the project that he considers to be closest to achieving exactly half the marks intended for this project, and will therefore receive the project mark of 10 points. The marks for all other submissions are linearly interpolated from those two fixed points, using the score from the test run as the weight of this linear interpolation.