Scrabble scoring -- Preliminary Functions

Write the following functions:

- `letterScore(letter, scorelist)` takes as input a single letter string called `letter` and a list where each element in that list is itself a list of the form `[character, value]` where `character` is a single letter and `value` is a number associated with that letter (e.g. it's scrabble score). The `letterScore` function then returns a single number, namely the value associated with the given `letter`. For example, you can cut and paste the following scrabble score list into your `hw2.py` program file. This function is allowed to crash if the `letter` is not in the `scorelist`.

```
scrabbleScores = [ ["a", 1], ["b", 3], ["c", 3], ["d", 2], ["e", 1],
["f", 4], ["g", 2], ["h", 4], ["i", 1], ["j", 8], ["k", 5], ["l", 1],
["m", 3], ["n", 1], ["o", 1], ["p", 3], ["q", 10], ["r", 1], ["s", 1],
["t", 1], ["u", 1], ["v", 4], ["w", 4], ["x", 8], ["y", 4], ["z", 10] ]
```

  If you include this in your file (outside of any function you define - for example right after the header comments in your file), then `scrabbleScores` is a "global variable"; it can be referred to by any function defined in that file and, more importantly for this example, it can be used once we load in that file.

```
>>> letterScore("c", scrabbleScores)
3
>>> letterScore("a", scrabbleScores)
1
```

- `wordScore(S, scorelist)` should take as input a string `S` and a `scorelist` in the format described above, which will have only lowercase letters, and should return as output the scrabble score of that string.
  Here are some examples:

```
>>> wordScore('spam', scrabbleScores)
8
>>> wordScore("wow", [['o', 10], ['w', 42]])
94
```

Scrabble Scoring

Your ultimate task here is to write a function that takes as input a "rack" - a **list** of letters - and returns the highest scoring word that can be made with those letters. This is the key ingredient in a computerized Scrabble game! Note about Scrabble: in this game, each letter in the rack can only be used once. For example, if the rack is ['w', 'y', 'l', 'e', 'l', 'o'] then you cannot make the word 'wow' because the rack only has one 'w'.

In this problem, you may use recursion as well as the built-in higher-order functions `map`, `filter`, and `reduce`. **In fact**, if you write a function that needs to process a long list (e.g. a dictionary) with more than a few hundred items in it, you are much better off letting `map`, `filter`, or `reduce` cruise through those lists since they have been optimized to work very fast.

There will be a few places where using anonymous functions (`lambda`) will be convenient.

One of the objectives of this problem is to have you think about designing a more complicated program that involves several functions. You have complete autonomy in deciding what functions to write in order to reach the ultimate goal (see more on the two required functions below). Try to think carefully about which functions you will need and try to implement those functions so that they are as simple and clean as possible. Don't settle for the first thing you think of. Strive to make your program simple and well organized.

Our sample solution has fewer than 9 functions, two of which we copied from our solution to the Recursion Muscles lab. The remaining 7 (or so) functions range from one to four lines of code per function. You are not required to have the same number of functions and you may have a few slightly longer function; the point is that it's possible to complete this assignment without writing a lot of code.

Be sure to have a comment at the top of the file with your name(s), the filename, the date, and the pledge. Every function should have a docstring saying what it does. If the function is even a little complicated, write a short comment explaining to the reader how this function works.

A bit later in this problem, we'll give you a fairly large dictionary of English words to use. For now, we recommend that you use the following tiny dictionary during the course of testing and development. Include this line near the top of your file. Now, it's a global variable that can be used by any of your functions in that file. Similarly, you should include the scrabble letter score list from Recursion Muscles.

```
Dictionary = ["a", "am", "at", "apple", "bat", "bar", "babble", "can", "foo",
"spam", "spammy", "zzyzva"]
```

Don't try to change the values of these global variables! As we'll see soon, that can make Python angry. However, there's no need to change the contents of the `Dictionary` or the scrabble letter scores during execution of the program.

The details...

Ultimately, there are two functions that we will be testing.

- `scoreList(Rack)` takes as input a `Rack` which is a list of lower-case letters and returns a list of all of the words in the global `Dictionary` that can be made from those letters and the score for each one. Specifically, this function returns a list of lists, each of which contains a string that can be made from the `Rack` and its Scrabble score. Here are some examples using the tiny `Dictionary` above:

```
>>> scoreList(["a", "s", "m", "t", "p"])

[['a', 1], ['am', 4], ['at', 2], ['spam', 8]]

>>> scoreList(["a", "s", "m", "o", "f", "o"])

[['a', 1], ['am', 4], ['foo', 6]]
```

The order in which the words are presented is not important.

- `bestWord(Rack)` takes as input a `Rack` as above and returns a list with two elements: the highest possible scoring word from that `Rack` followed by its score. If there are ties, they can be broken arbitrarily. Here is an example, again using the tiny `Dictionary` above:

```
>>> bestWord(["a", "s", "m", "t", "p"])

['spam', 8]
```

Aside from these two functions, all of the other helper functions are up to you! Some of those helper functions may be functions that you wrote in Recursion Muscles, or slight variants of those functions. However, you might find it useful to use a strategy in which you write a function that determines whether or not a given string can be made from the given `Rack` list. Then, another function can use that function to determine the list of *all* strings in the dictionary that can be made from the given `Rack`. Finally, another function might score those words.

Remember to use `map`, `reduce`, or `filter` where appropriate - they are powerful and they are optimized to be very fast.

Test each function carefully with small test inputs before you proceed to write the next function. This will save you a lot of time and aggravation!

A reminder about `in`: Imagine that you are writing a function that attempts to determine if a given string `S` can be made from the letters in the `Rack` list. You might be tempted to scramble ("permute" to use a technical term) the `Rack` in every possible way as part of this process, but that is more work

than necessary. Instead, you can test if the first symbol in your string, `S[0]`, appears in the rack with the statement:

```
if S[0] in Rack:

  # if True you will end up here

else:

  # if False you will end up here
```

That `in` feature is very handy. Now, recursion will let you do the rest of the work without much effort on your part!

Although we don't expect that you will end up doing huge recursive calls, you should know that Python can get snippy when there are a lot of recursive calls. By default, Python generally complains when there are 1000 or more recursive calls of the same function. To convince Python to be friendlier, you can use the following at the top of your file.

```
import sys

sys.setrecursionlimit(10000)  # Allows up to 10000 recursive calls
```

Trying it with a bigger dictionary

Finally, if you want to test out a big dictionary (it's more fun than the little one above), you can use the file `dict.py` provided with the assignment. It contains a large list of words (a bit over 4000 words). The list is still called `Dictionary`. After you have downloaded the file and placed it in the same directory as your Python file, you should do the following at the top of your Python file:

```
from dict import *
```

Now, you can refer to `Dictionary` as a global variable. This is much nicer than cutting-and-pasting that very large list into your Python file. Here are some examples using that `Dictionary`.

```
>>> scoreList(['w', 'y', 'l', 'e', 'l', 'o'])

[['leo', 3], ['low', 6], ['lowly', 11], ['ow', 5], ['owe', 6], ['owl', 6],
['we', 5], ['well', 7], ['woe', 6], ['yell', 7], ['yo', 5]]
```

Notice that "yellow" is not in this dictionary. We "cropped" off words of length 6 or more to keep the dictionary from getting too large. Finally, here is an example of `bestWord` in action:

```
>>> bestWord(['w', 'y', 'l', 'e', 'l', 'o'])

['lowly', 11]
```

```
>>> bestWord(["s", "p", "a", "m", "y"])

['may', 8]

>>> bestWord(["s", "p", "a", "m", "y", "z"])

['zap', 14]
```

And an even bigger dictionary!

Want to try this with an even bigger dictionary? We've got one! It's in the file called `bigdict.py` provided with the assignment. The bad news is that it causes IDLE to crash on some computers, but you can always re-start IDLE and go back to a smaller dictionary.

If you want to try this, download the file and use this:

```
from bigdict import *
```

The dictionary is still called `Dictionary` and it is available as a global variable. Here is an example of it in use:

```
>>> scoreList(['a', 'b', 'v', 'x', 'y', 'y', 'z', 'z', 'z'])

[['ab', 4], ['aby', 8], ['ax', 9], ['ay', 5], ['ba', 4], ['bay', 8], ['by',
7], ['ya', 5], ['yay', 9], ['za', 11], ['zax', 19], ['zyzzyva', 43], ['zzz',
30]]
```