6.00 Introduction to Computer Science and Programming
Fall 2008

6.00: Introduction to Computer Science and Programming

Problem Set 6: Word Game II

Handed out: Thursday, October 9th, 2008.
DUE: **11:59pm Thursday October 16th, 2008**

Introduction

In this problem set you will write a program that will play the 6.00 word game all by itself. It is an extension to Problem Set 5, in which you wrote a word game that a human could play.

Workload

Please let us know how long you spend on each problem. We want to be careful not to overload you by giving out problems that take longer than we anticipated.

Collaboration

You may work with other students. However, each student should write up and hand in his or her assignment separately. *Be sure to indicate with whom you have worked.* For further details, please review the collaboration policy as stated in the syllabus.

Problem #1: How long?

You have a friend who consistently beats you when playing the word game because she takes forever to play. You decide to change the rules of the game to fix her wagon. Points are awarded as before, except the points awarded for a word are divided by the amount of time taken to find the word. Points for a word should be displayed to two decimal places.

First, make a copy of your `ps5.py` and call it `ps6.py`
(Make sure the word list, words.txt, is also in the same directory.)

Modify the `play_hand` function so that your game looks something like this (note the lines in bold):

```
Current Hand:  a c i h m m z
Enter word, or a . to indicate that you are finished: him
It took 6.47 seconds to provide an answer.
him earned 1.24 points. Total: 1.24 points
Current Hand:  a c m z
Enter word, or a . to indicate that you are finished: cam
It took 3.25 seconds to provide an answer.
cam earned 2.15 points. Total: 3.39 points
Current Hand:  z
Enter word, or a . to indicate that you are finished: .
Total score: 3.39 points.
```

Note: What happens when someone enters a word extremely fast? Because time is calculated in discrete chunks with some minimum value, it is possible to enter a word so fast that the total time taken rounds to zero seconds and a zero divide error occurs. If this happens, decide what you think is a reasonable thing to do, and do it. Be sure to document your change.

Hint: The following code will tell you how long it takes to enter your name:

```
import time

start_time = time.time()
```

```
name = raw_input('What is your name?: ')
end_time = time.time()
total_time = end_time - start_time
print 'It took %0.2f to enter your name' % total_time
```

Problem #2: Time Limit

You still find it boring to watch your friend think for long periods of time while playing the 6.00 word game. You decide to add a "chess clock" to the game. This limits the total amount of time, in seconds, that a player can spend to play a hand.

Modify your game so the output looks something like the following. Your program should prompt for a time limit and stop counting score for any words entered after the time has run out. In addition, print the time remaining (if it is >=0) after each input.

You should time the user input for **both** valid and invalid words and include them in the time total. Also, you should time **only** the user input, and not the processing done by the computer.

```
Enter time limit, in seconds, for players: 8
Current Hand:  a c i h m m z
Enter word, or a . to indicate that you are finished: him
It took 6.47 seconds to provide an answer.
You have 1.53 seconds remaining.
him earned 1.24 points. Total: 1.24 points
Current Hand:  a c m z
Enter word, or a . to indicate that you are finished: cam
It took 3.25 seconds to provide an answer.
Total time exceeds 8 seconds. You scored 1.24 points.
```

Problem #3: Computer Player

You've spent so much time working on 6.00 that, unfortunately, you don't actually have any friends left to play the word game with you. So, nerd that you have become, you decide to implement support for computer players.

Instead of having the user enter words, we will replace the call to `raw_input` (inside `play_hand`) with a call to the following function:

```
def pick_best_word(hand, points_dict):
    """
    Return the highest scoring word from points_dict that can be made with the
    given hand.

    Return '.' if no words can be made with the given hand.
    """
    ...implement me!...
```

(Some of you may object to the name of this function. The highest scoring single word may not be the best in terms of maximizing the total value that can be extracted from a hand. We will talk about this issue later in the term.)

As the first step of this problem, write the following pre-processing function:

```
def get_words_to_points(word_list):
    """
    Return a dict that maps every word in word_list to its point value.
```

```
    """
```

Think about how many times this function needs to be called -- should it be called once per game, once per hand, or once per turn within a single hand? For this implementation, you should create a global variable `points_dict` to store the resulting dictionary.

Next, change the implementation of `is_valid_word` to take as an argument the representation you created above rather than the `word_list` itself. Remember that the choice of representation can have a big impact on performance. This modification will improve the complexity of `is_valid_word` from O(`len(word_list)`) to O(`1`).

Finally, write `pick_best_word`. (There's no need to be overly clever or worry about optimizing it a lot; write a straightforward implementation that you can easily debug.) Modify `play_hand` to call it.

It wouldn't be fair to give the computer player the same time that is allowed to human players. Use the following function to set the time limit for your computer. Notice that it is intended to deal with the fact that some computers are faster than others by timing some basic operations. Test your implementation using k=1 and other values.

```
import time

def get_time_limit(points_dict, k):
    """
    Return the time limit for the computer player as a function of the
    multiplier k.

    points_dict should be the same dictionary that is created by
    get_words_to_points.
    """
    start_time = time.time()
    # Do some computation. The only purpose of the computation is so we can
    # figure out how long your computer takes to perform a known task.
    for word in points_dict:
        get_frequency_dict(word)
        get_word_score(word, HAND_SIZE)
    end_time = time.time()
    return (end_time - start_time) * k
```

Again, think of how often this function should be called. Define a global variable `time_limit` to store the result of this function.

Update on 13 Oct: Changed the function signature to avoid confusion between `word_dict` and `points_dict`. Note that `points_dict` is the one that should be passed to both `get_time_limit` and `pick_best_word`.

Problem #4: Even Faster Computer Player

Now implement a faster computer player called `pick_best_word_faster(hand, rearrange_dict)`. It should be based on the following approach described below. (This is a good example of what **pseudocode** should look like).

First, do this pre-processing before the game begins:

```
Let d = {}
For every word w in the word list:
    Let d[(string containing the letters of w in sorted order)] = w
```

After the above pre-processing step, you have a dict where, for any set of letters, you can determine if

there is some acceptable word that is a **rearrangement** of those letters. You should put the pre-processing code into a separate function called `get_word_rearrangements`, analagous to `get_words_to_points`.

As in Problem 3, decide where this function should be called based on how many times it needs to run. Store the returned value in a global variable `rearrange_dict`.

Now, given a hand, here's how to use that dict to find a word that can be made from that hand:

```
To find some word that can be made out of the letters in HAND:
    For each subset S of the letters of HAND:
        Let w = (string containing the letters of S in sorted order)
        If w in d: return d[w]
```

N.B.: These are actually *sub-multisets*, not subsets. In a formal definition, sets cannot contain repeated elements, while multisets, like groups of letters within a hand, can.

Create `pick_best_word_faster` based on the pseudocode above. Modify it to return not just any valid word (as described above) but the **highest-scoring word** that can be made out of the letters in `hand`.

Then modify your `play_hand` so that it calls `pick_best_word_faster` rather than `pick_best_word`.

Update on 13 Oct: Changed the function signature to avoid confusion between `word_dict` and `rearrange_dict`. Note that `rearrange_dict` is the one that should be passed to `pick_best_word_faster`.

Problem #5: Algorithm Analysis

Characterize the time complexity of your implementation (in terms of the size of word_list and the number of letters in a hand) of both `pick_best_word` and `pick_best_word_faster`.

Please put your response in comments at the end of `ps6.py`, like so:

```
## Problem 5 ##
# your response here.
# as many lines as you want.
```

Handin Procedure

1. Save

All your code should be in a single file called `ps6.py`. This file should include complete and tested implementations of `get_words_to_points`, `get_word_rearrangements`, `pick_best_word`, and `pick_best_word_faster`. In your final `ps6.py` submission, the functions `play_game` and `play_hand` should play the 6.00 word game using your choice of `pick_best_word` or `pick_best_word_faster`.

2. Time and collaboration info

At the start of the file, in a comment, write down the number of hours (roughly) you spent on this problem set, and the names of whomever you collaborated with. For example:

```
# Problem Set 6
```

```
# Name: Jane Lee
# Collaborators: John Doe
# Time: 1:30
#
 .... your code goes here ...
```