CSE 341, Spring 2008, Assignment 7
Due: Friday 6 June, 8:00AM

Last updated: May 24

Overview: You will use Ruby to implement a very simple game where the program prints out a sequence of letters and the player types in a subset of the letters, which are then replaced to produce a new sequence of letters. A player's score increases as the game proceeds. The game ends when the player enters a blank line or a sequence of letters that is not a subset of the current letters printed by the game. The course website has some sample interactions of the game being played. More specific rules about how the game works are in problem 3.

You will use some object-oriented programming techniques to make the game configurable in how it generates letters and how it keeps score. Style matters, probably more than on previous assignments. The sample solution is about 115 lines, but since this assignment gives less specific direction than previous assignments, your solution might be moderately shorter or longer, but do not write an excessively long solution. You may want to use methods on arrays, strings, or files that we have not seen in class, but you have seen how to browse the standard libraries.

1. In this problem, you will define two subclasses of this class, which you should type into your solution:

```
class GetRandomLetter
  def get_letter
    r = rand
    @counts.each_index {|i| if r < @counts[i] then return (i+97).chr end }
  end
end
```

   The purpose of `get_letter` is to return a string holding one lower-case letter. Do not add any methods to `GetRandomLetter` and do not override `get_letter` in your subclasses. Note that part of the problem is understanding how `get_letter` works and what it assumes about subclassess. Hint: It has something to do with *cumulative probability*. Hint: `rand` is a method defined in `Object` and there is no need to override it.

   (a) Define a class `GetRandomLetterUniform` for which `get_letter` returns each of `"a"`, `"b"`, ..., `"z"` with equal probability (i.e., 1.0/26.0). Its constructor needs no arguments.

   (b) Define a class `GetRandomLetterFile` with a constructor that takes one argument, a string that is a file name. It uses this file to calculate the probability of returning each letter lower-case letter as follows: If p (e.g., .23) of the lowercase letters in the file are character x (e.g., `c`), then `get_letter` will return x with probability p.

2. In this problem, you will define four classes that have a method `score`, which takes a string and returns an integer (the number of "points" you get in the game for using the string).

   (a) Instances of `ScoreWordSize` should just return the length of the word.

   (b) Instance of `ScoreWordLongest` should return the length of the word unless the word is the longest word that has ever been scored by this object. In that case, the score should be the word's length plus 10.

   (c) Instances of `ScoreWordSizeUnique` are like `ScoreWordSize` except they give a score of 0 if the same word has been previously scored by this object.

   (d) Instances of `ScoreWordLongestUnique` are like `ScoreWordLongest` except they give a score of 0 if the same word has been previously scored by this object.

   Define one mixin so that the bodies of the class definitions for `ScoreWordSizeUnique` and `ScoreWordLongestUnique` simply include the mixin.

3. Write a top-level method `rungame` that runs the game.

   - `rungame` takes 3 arguments, all of which have default values (so any suffix of them can be omitted):
     - The first argument is an integer, which is how many letters the player is "given" at a time. We call the letters the player has the "rack", so this argument is the length of the rack. Default value is 7.
     - The second argument is an object with a `score` method that takes a string and returns a number. Default value is `ScoreWordSize.new`. We call this argument the scorer.
     - The third argument is an object with a `get_letter` method that takes no arguments and returns a lower-case letter in a string. Default value is `GetRandomLetterFile.new "/usr/share/dict/words"`. We call this argument the `lettermaker`.
   - The game repeatedly prints a line like this and then waits for the player to input a string.

     ```
     Current letters: ivphuhm, Score: 0
     ```

     where in general the string before the comma is the current contents of the rack and the number is the current score. The score starts at 0. The initial rack is created by using the lettermaker racklength times. Note the rack may contain repeated letters.
   - If the player enters the line `"\n"` (i.e., the empty string plus the newline character always added by `gets`), the game is over and `rungame` should simply return `true`.
   - If the player enters a line that cannot be formed by a subset of the letters in the rack, the game is over and `rungame` should return `false`. If the player uses a character n times, then that character must be in the rack at least n times. Note there is no requirement that the player enters a "word" — any sequence of letters that is a subset of the rack is acceptable. Note you should remove the `"\n"`; we do not consider that part of the player's letter sequence.
   - The score is increased by using the amount the scorer gives the letter sequence. The score may increase by 0.
   - Before the player enters another letter sequence, the rack is updated by removing letters from the rack that are in the letter sequence and replacing them with letters produced by the lettermaker. If the player's letter sequence uses a character n times, then exactly n occurrences of the character are replaced in the rack. Note there is some probability a character will be "replaced" by the same character.

4. **Challenge Problem:** *Make up your own challenge problem.* It should involve writing a moderate amount of additional Ruby code, be related to the game, and demonstrate interesting object-oriented programming techniques. In addition to writing the code, you must write the question. That is, in a comment in your Ruby code write a "homework assignment problem" for which your code is a good solution. Your question must be well-writen, concise, and precise.[1]

Turn-in Instructions

- Put all your solutions in one file, `lastname_hw7.rb`, where `lastname` is replaced with your last name.
- The first line of your `.rb` file should be a Ruby comment with your name and the phrase `homework 7`.
- Go to `https://catalysttools.washington.edu/collectit/dropbox/djg7/2125` (link available from the course website), follow the "Homework 7" link, and upload your file.

[1] This may help you appreciate that writing homework specifications is not easy.