Bulls and Cows with Words

(DRAFT VERSION TO BE FINALIZED IN MARCH 22, 2020)

Rules of the game

Bulls and Cows is a classic British pen-and-paper game for two players. One player tries to guess the secret number chosen by the other player with no digit appearing more than once, for example 8451. The guesser submits his guesses for the secret number one guess at the time. After each guess, the opponent reveals how many **bulls** (right digit in the right position) and **cows** (right digit but in a wrong position) that guess contains. For example, the guess 7415 contains one bull (the digit 4) and two cows (the digits 1 and 5), whereas the guess 9257 contains one bull (the digit 5) and zero cows. Each guess therefore provides more information about the secret number by eliminating a bunch of numbers that were still possible after the previous guesses.

This project adapts the basic rules of this game from integers to the more interesting domain of English words composed of the lowercase letters `a-z`, as listed in the file `words_alpha.txt`, to create a whole new game that resembles another classic game of Jotto. Played with the set of English words in which each letter occurs at most once, the restricted structure of the language and its statistical letter combinations allow the guesses to be made in ways that have a higher chance of guessing the secret word earlier, compared to blindly making random guesses among the words that are still possible, given the information gained from the previous guesses.

The instructors provide the automated tester scripts `bacrunner.py` (for Python 3) or `BACRunner.java` (for Java 8) that act as the opponent and scorekeeper for the game. The game is played for several rounds to diminish the effect of blind random luck. The total number of guesses made over all these rounds becomes the final score of the player. As in golf, the lower the score, the better.

(For anybody who has been pleasantly infected with a recreational linguistics bug, the longest word in the English language that has no repeated letters seems to be `dermatoglyphics`. For those interested in the combinatorial mathematics of the original four-digit integer version of the game, five guesses always suffice to uniquely pinpoint the secret number, provided that each guess is made systematically based on all of the available information from the previous guesses. But we don't need such advanced combinatorics to play this game, although such ideas can help improve the guessing accuracy of your player once its basic mechanics are working.)

Required functions / methods

The student must implement the player into the script `bacplayer.py` if working in Python 3, and in the file `BACImplementation.java` if working in Java 8. Inside these files, the student can write whatever functions to wish to implement the guessing logic, but to allow the tester script to seamlessly interact with the player, the player script **must** define the following functions into its namespace, each function with the signature exactly as specified below. The Java player class must `implement BACPlayer` to promise the existence of the following methods.

```
def author(): # Python
public String getAuthor(); // Java
```

Returns the name of the author of this player as a string in the form `"Last, First"`, exactly as your name appears in RAMMS.

```
def student_id(): # Python
public String getStudentID(); // Java
```

Returns the student ID of the author as a string without spaces, such as `"123456789"`.

```
def initialize_wordlist(words): # Python
public void initializeWordList(List<String> words); // Java
```

The tester script calls this function exactly once in the beginning, before commencing the play of the individual rounds. The parameter `words` is the list of English words that could appear as secret words in the game, guaranteed to be in sorted lexicographic order. This wordlist is not passed as an argument to the other functions later in the game, so the player script is supposed to store and remember the wordlist for later use at this stage. This function is also welcome to perform any preprocessing of the wordlist and organizing its content into any data structures of your choosing that might later improve the speed and accuracy of guessing.

(Inside this Python function, the keyword `global` that allows the function statements to create and access a name outside the function body will probably come handy.)

```
def guess(n, guesses): # Python
public String guess(int n, List<String> guesses, List<Integer> bulls,
List<Integer> cows); // Java
```

The heart and soul of this project. Given the length of the secret word `n`, and the list of `guesses` the player has done so far to reveal the current secret word (this argument list is therefore empty

when guessing the secret word for the first time), this function should return the word that the player wishes to be the next guess.

In Python, each individual guess in the `guesses` list is guaranteed to be a tuple `(word, bulls, cows)` for each `word` that the player has guessed so far, along with its revealed number of `bulls` and `cows`. For example, if the `guesses` argument is `[('washmen', 0, 3), ('luiseno', 1, 2), ('thorina', 1, 3)]`, some computation reveals that exactly 27 words from `words_alpha.txt` still remain as possible secret words consistent with these counts. Further guessing is therefore needed to pare down the remaining possibilities until only one word remains. (However, not all guesses are equally useful in how much information they tend to produce the times that they miss for various counts for bulls and cows...)

In Java, due to the lamentable lack of tuples and triples in that language, the `(word, bulls, cows)` information triple is split into three separate lists so that the first list `guesses` contains the list of the previously guessed words. For each `word`, the corresponding element in the list `bulls` is the number of bulls in that same word, and similarly for `cows`.

Runner script parameters

The `bacrunner.py` script first defines the variables `minlen` and `maxlen` for the minimum and maximum length of the secret word, thus ignoring all the words from `words_alpha.txt` whose length is outside these bounds. The variable `rounds` determines how many secret words the game is played for before tallying up the final score. The tester script takes care of filtering of the words that contain duplicated letters, so that the player script does not have to worry about any illegal words in the given wordlist.

The choice of random secret words is done in **pseudorandom** fashion using the `seed` value defined inside the `bacrunner.py` script. Using a fixed `seed` value, the series of secret words produced by the tester will always be the same in every run. The fact that once the `seed` value has been fixed, all seemingly possible outcomes of the potentially branching future are illusory because the resulting execution is in reality just as deterministic as a train destined to follow its tracks, is actually a very good thing during the development stage! When debugging your script or optimizing its logic and accuracy, the exact same series of secret words can be repeated as often as you need to compare the results of the script before and after each change.

One example run for ten rounds using one of the author's private model solutions produced the following output. After printing out the current parameters, each line first displays the secret word in square brackets, followed by the incorrect guesses made before hitting the secret word. For example, this particular player implementation (this version always starts from randomly chosen first word from the list of all words of the given length, which is not the optimal approach) needed five guesses to hit the secret word `blinkers`. (Curiously enough, the longer

the secret word, the fewer guesses it usually takes to hit!) The last line of the output displays the total score, followed by the student ID and name information of that player.

```
BACRunner (Python) with seed=888.
[blinkers]: geckotid symploce hayricks poniards blinkers
[larkish]: placebo skywave updives bashkir britska larkish
[plastin]: mudbank recombs paulite plastin
[linters]: forsake punkies linters
[motives]: relishy isogamy cagiest vetoism motives
[indra]: slait beira brims incra indra
[rashti]: truced baetyl palour harems maigre rashti
[hunts]: moity hocks doles hafts hurts hunts
[overawful]: septiform adverting psychidae kingbolts overawful
[kingcraft]: obscurant fostering discovert kingcraft
49 123456789 Kokkarinen, Ilkka
```

Grading

For the grading of this project, the instructors will use a secret `seed` value of their own. The exact same `seed` value will be used to grade all submitted projects, so that the final scores will be directly comparable to each other. Submitted solutions will be tested for **100 rounds of secret words** to diminish the effect of luck (statistically proportional to the square root of `rounds`) relative to the effect of skill (statistically linearly proportional to `rounds`). For example, during the guessing of `hunts` in the above example run, after the first four failed guesses both words `hurts` and `hunts` are equally likely to be the secret word, so choosing between these two is a genuine coin toss. However, making better choices during the earlier guesses while the list of possible secret words is longer will tend to nudge the luck to go to your direction more often than it will nudge it away from you.

To prevent some malfunctioning player from getting hopelessly stuck so that some secret word will never get guessed, **a mercy rule is in effect** inside the tester script so that after twenty wrong guesses, the current secret word is considered to have been correctly guessed, and the game moves on to the next round of a new secret word. However, please note that this mercy rule cannot prevent the player script from getting internally stuck in an infinite loop. Therefore, before submitting your `bacplayer.py` file on D2L for grading, please make it run overnight for some large number of `rounds`, at least in the thousands. This will effectively execute what is known in this line of work as a **fuzzed stress test** to hammer your code and force it to reveal all its internal fractures and other weaknesses. If the execution has terminated and the last line of the output displays your information and total score to proudly greet you as you get up the next morning, your `bacplayer.py` ought to be sturdy enough to be submitted with confidence.

In all fields of engineering both virtual and physical when dealing with a new problem, it is usually easy to gain big morale-boosting improvements in the initial stages of both design and implementation. In this project, relatively simple algorithmic and heuristic improvements will see the total number of guesses made by your player fall by leaps and bounds. However, once the low-hanging branches have been picked bare of all their juicy fruit and you have to climb ever higher towards the theoretical asymptotic optimum of the problem that is fundamentally determined by the laws of reality that the problem is embedded in, each successive improvement will always gain less and yet always be more difficult to achieve! Finally at the unknown optimum lower bound, even the greatest data scientist could not conjure information to get through that rock hard limit any more than it would possible to exceed the speed of light.

(As a helpful analogy for this phenomenon of "the last mile", imagine the vastly different technologies required to remove 90% of dust particles from the air currently inside the given room, as opposed to the goal of removing 99% of these particles, let alone 99.999% to operate an industrial-strength chip manufacturing cleanroom. Or, consider the increasing engineering challenges of trying to build an automobile that can achieve a top speed of 100, 200 or 300 kilometers per hour.)