Evil Hangman

Evil Hangman Algorithm Analysis:

Evil Hangman is quite like Hangman, except that the computer cheats. Rather than pick, say, a seven-letter word at the game's start, the computer instead starts off with a mental list (well, maybe a set or array) of all possible seven-letter English words. Each time the human guesses a letter, the computer checks whether there are more words in its list with the letter than without and then whittles the list down to the largest such subset. If there were more words with the letter than without, the computer congratulates the user and reveals the letter's location(s). If there were more words without the letter than with, the computer just laughs.
Put more simply, the computer constantly tries to dodge the user's guesses by changing the word it's (supposedly) thinking of. Pretty evil, huh? The funny thing is most humans wouldn't catch on to this scheme. They'd instead conclude they're pretty bad at Hangman. Mwah hah hah.
Suffice it to say that the challenge ahead if you is to implement... Wait for it...Evil Hangman.

▪But what is the algorithm for evil? Well, let's consider what the computer needs to do anytime the user guesses a letter. Suppose that the user needs to guess a four-letter word and that there are only a few such words in the whole English language: `BEAR`, `BOAR`, `DEER`, `DUCK`, and `HARE`. And so the computer starts off with a universe (*i.e.*, list) of five words. Next suppose that the user guesses `E`. A few of those words contain `E`, so the computer had best decide how to dodge this guess best. Let's partition those words into "equivalence classes" (a la CSCI E-207) based on whether and where they contain `E`. It turns out there are four such classes in this case:
`----`, which contains `BOAR` and `DUCK`
`-E--`, which contains `BEAR`
`-EE-`, which contains `DEER`
`---E`, which contains `HARE`
Note that `HARE` is not in the same equivalence class as `BEAR`. Even though both have just one `E`, each's `E` is in a different location. If the computer is forced to admit that the word that it's "thinking of" contains an `E` (because it just so happens to have discarded all words that lack `E` in response to the

user's past guesses), it will be forced to announce (and thus commit to) that letter's location.
So, back to our story: the user has guessed `E`. What should the computer now do? Well, it could certainly declare that the word it's "thinking of" does not contain `E`, the implication of which is that the list of five words becomes two (`BOAR` and `DUCK`). That does feel optimal: armed with two words, the computer might still have a chance to "change its mind" yet again later. Then again, `DUCK` seems pretty easy to guess, whereas most people might not even think of `HARE`. On the other hand, I don't remember the last time I used `BOAR` in a sentence. But let's keep it simple: you may assume that the computer will always react to a user's guess by whittling its list down to the largest equivalence class, with pseudorandomness breaking any ties.

Realize, though, that the largest equivalence class might not always correspond to `----`. For instance, suppose that the user had guessed `R` instead of `E`. The equivalence classes would thus be as follows:

`----`, which contains `DUCK`
`---R`, which contains `BEAR`, `BOAR`, and `DEER` `--R-`, which contains `HARE`

In this case, the computer should go ahead and admit that the word that it's "thinking of" contains `R` at its end, since that leaves three possible words (`BEAR`, `BOAR`, and `DEER`) and thus the maximum amount of maneuverability down the road in reaction to subsequent guesses. Of course, if the user has never heard of a `DUCK`, then `----` could very well be a superior strategy. But, again, lest you drive yourself nuts with overanalysis, you may assume that the largest equivalence class is optimal, even though it might not be in reality.

Indeed, that strategy might sometimes backfire, at least in a sense. Suppose that, in a new version of the story at hand, there are only three four-letter words in the English language: `BOAR`, `DEER`, and `HARE`. Suppose now that the user guesses `E`. In this case, our equivalence classes are:

`----`, which contains `BOAR` `-EE-`, which contains `DEER` `---E`, which contains `HARE`

Because each has one word, these classes are, of course, of the same size. But if we happen to select `-EE-` pseudorandomly, thereby whittling our list down to just `DEER`, we'll have revealed to the user two of the word's letters, whereas we could have revealed zero (had we selected `----` instead) or one (had we selected `---E` instead). But, again, that's okay. You are welcome, but not required, to implement a more sophisticated algorithm than that prescribed here; just make clear in some comments how yours happens to work.

Specifications:

Your challenge for this project is to implement Evil Hangman as a native iOS app. The overall design and aesthetics of this app are ultimately up to you, but we require that your app meet some requirements. **All other details are left to your own creativity and interpretation.**

Features:

▪Immediately upon launch, gameplay must start (unless the app was simply backgrounded, in which case gameplay, if in progress prior to backgrounding, should resume).

▪Your app's front side must display placeholders (*e.g.*, hyphens) for yet-unguessed letters that make clear the word's length.

▪Your app's front side must inform the user (either numerically or graphically) how many incorrect guesses he or she can still make before losing.

▪Your app's front side must somehow indicate to the user which letters he or she has (or, if you prefer, hasn't) guessed yet.

▪The user must be able to input guesses via an on-screen keyboard.

▪Your app must only accept as valid input single alphabetical characters (case-insensitively).
Invalid input (*e.g.*, multiple characters, no characters, characters already inputted, punctuation, *etc*.) should be ignored (silently or with some sort of alert) but not penalized.

▪Your app's front side must have a title (*e.g.*, **Hangman**) or logo as well as two buttons: one that flips the UI around to the app's flipside, the other of which starts a new game.

▪If the user guesses every letter in some word before running out of chances, he or she should be somehow congratulated, and gameplay should end (*i.e.*, the game should ignore any subsequent keyboard input). If the user fails to guess every letter in some word before running out of chances, he or she should be somehow consoled, and gameplay should end. The front side's two buttons should continue to operate.

▪On your app's flipside, a user must be able to configure two settings: the length of words to be guessed (the allowed range for which must be [1, *n*], where *n* is the length of the longest word in `words.plist`); and the maximum number of incorrect guesses allowed (the allowed range for which must be [1, 26]).

▪When settings are changed, they should only take effect for new games, not one already in progress, if any.

Implementation Details:

▪Your app's UI should be sized for an iPhone or iPod touch (*i.e.*, 320 × 480 points) with support for, at least, `UIInterfaceOrientationPortrait`. However, if you or your partner owns an iPad and would prefer to optimize your app for it (*i.e.*, 768 × 1024 points), you may, so long as you inform your TF prior to this project's deadline.

▪You must use the contents of `words.plist` as your universe of possible words. You're welcome, but not required, to transform it into some other format (*e.g.*, SQLite).

▪Your app must come with default values for the flipside's two settings; those defaults should be set in `NSUserDefaults` with `registerDefaults:`. Anytime the user changes those settings, the new values should be stored immediately in `NSUserDefaults` (so that changes are not lost if the application is terminated).

▪You must implement each of the flipside's numeric settings with a `UISlider`. Each slider should be accompanied by at least one `UILabel` that reports its current value (as an integer).

▪You are welcome to implement your UI with Xcode's interface builder in `MainViewController.xib` and `FlipsideViewController.xib`, or you may implement your UI in code in `MainViewController.{h,m}` and `FlipsideViewController.{h,m}`.

▪You must obtain a user's guesses via a `UITextField` (and the on-screen keyboard that accompanies it). For the sake of aesthetics, you are welcome, but not required, to keep that `UITextField` hidden (so long as the on-screen keyboard works). You are also welcome, but not required, to respond to user's keypresses instantly, without waiting for them to hit **return** or the like, in which case `textField:shouldChangeCharactersInRange:replacementString` in the `UITextFieldDelegate` protocol might be of some interest.

▪Your app must use Automatic Reference Counting (ARC).

▪Your app must tolerate low-memory warnings (as by reloading views when needed).