CPSC 231 ASSIGNMENT #4

Hangman

Hangman is a classic two-player guessing game, played on pencil and paper, that most of us probably learned in elementary school. One player thinks of a "secret" word, and the other player's goal is to try to determine the secret word by guessing its letters one at a time.

Your main objective in this assignment is to write a computer program that will play a game of Hangman with you. Your program will come up with the secret word and keep track of your guesses, and you, or anyone playing your game on the computer, will be the one guessing the word.

The rules for our game of Hangman are as follows. The computer will first choose a random secret word that is at least four letters long from a lexicon of the 4000 most frequently used English words. It shows the word first as blanks (underscores or dashes), with one blank for each letter of the word. Then play proceeds as follows:

1. The player chooses a letter of the alphabet and indicates it to the computer as his or her guess.
   - If the secret word contains the letter, all occurrences of the letter in the word are revealed.
   - Otherwise, the letter is written in the list of incorrect guesses, and an additional part of the stickman is drawn.
2. If all letters of the secret word are revealed, the game is won. If the gallows and all parts of the man have been drawn, the game is lost. Otherwise, play continues with step 1.

Our Hangman game will allow the human player a total of eight guesses to determine the secret word. Each incorrect guess will result in the drawing of the following parts, in this order: head, torso, left arm, right arm, left leg, right leg, gallows, face. The game is lost once the face is drawn. A game in progress is shown in Figure 1.

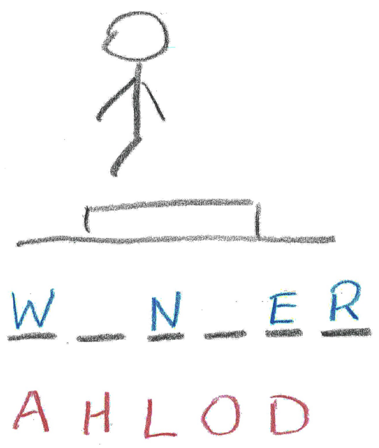

Figure 1: A game of Hangman in progress. Four letters of the secret word have been guessed correctly, and are revealed in blue. Five incorrect guesses have been made, written below in red.

Due Dates

Individual component	Friday, October 26, 11:59 PM
Paired component	Friday, November 2, 11:59 PM

Individual Component

These first two problems in the individual component of this assignment are designed to help you get familiar with reading a list of words from a file, performing computations on them, and drawing simple things to a window on your screen. You can use the exact same programming environment you set up for previous assignments to complete this assignment. Create and submit a separate Python program for each problem.

Problem 1: Word Lookup

Write a Python program that will read the provided lexicon[1] file and perform a lookup for a user-specified word in the lexicon. The lexicon file, `cpsc231-lexicon.txt`, is a text file that contains the 4000 most frequent words used in contemporary American English,[2] with one word written per line in order of their frequency of use. All words are written lowercase except for the word "I".

[1] A lexicon is like a dictionary, but without definitions for the words.

[2] Apologies that I was unable to find a lexicon of most common words in Canadian English.

Your program should first indicate at what rank position the query word appears in the list, or that the query word is not contained in the lexicon. Then it should output, in alphabetical order and without duplicates, a list of all the letters that occur in the query word.

Inputs: A lexicon file name specified as the first command line argument, and a query word as the second command line argument. For example, you might run your program with the following invocation:

```
$ python3 my-p1.py cpsc231-lexicon.txt color
```

Outputs: An indication of what frequency rank the query word appears in the lexicon, or that it was not found in the provided file. Then print a list of all letters contained within the query word, in alphabetical order. For example, your program's output may look like those shown on the right.

Example Problem 1 output:

```
According to our lexicon, "color"
is the 691st most common word
in contemporary American English.

It contains the following letters:
 c l o r
```

Another sample run:

```
According to our lexicon, "colour"
is not in the 4000 most common words
of contemporary American English.

It contains the following letters:
 c l o r u
```

Problem 2: Analog Clock

The `time` module in Python provides a convenient means for your program to ask for your computer's current local time. When you call the `localtime()` function in the `time` module, it gives you back an aggregate data structure[3] with named fields that tell you what time it is. For example, you can obtain the hour, minute, and second for the current time as follows:

[3] We'll learn more about these later in the course, but it's simple enough to get what you want by following exactly the syntax shown for now.

```
import time
time_aggregate = time.localtime()
hour = time_aggregate.tm_hour
```

```
    minute = time_aggregate.tm_min
    second = time_aggregate.tm_sec
```

Use this information to create a program that draws an analog clock displaying the time (hour, minute, and second) at which your program is run. Your clock should have a circular face with the numerals 1–12 to indicate the hour. It should also have an hour hand that is shorter and distinct in appearance from the minute hand, which is in turn shorter and distinct from the second hand. Apart from these basic requirements, you may make your clock appear as you like.

It may be neat to animate the clock as specified in Creative Exercise 1.5.32 of your text, though you are not required to do so for this problem. Sorry, doing this is not enough for a bonus either!

Inputs: None.

Outputs: An analog clock with hour, minute, and second hands showing the current time, drawn in a window on your screen. It should look similar to the one shown on the right.

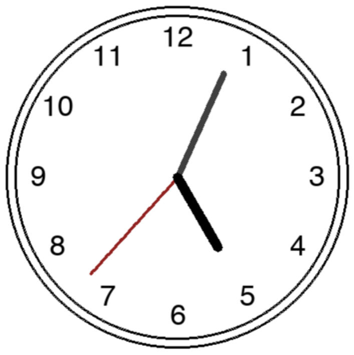

Figure 2: Example Problem 2 output.

Paired Component

We encourage you to work with a partner to complete the remainder of this assignment. You may choose your own partner, but remember that you must work with a *different partner* for each assignment in this class! If you are solving these problems as a pair, one submission is sufficient for both students.

Problem 3: Console Hangman

Your first goal for the paired component is to create an interactive game of Hangman that can be played entirely through the console window. Your program will play the role of "executioner", selecting the secret word, keeping track of letter guesses, and detecting when the game is won or lost.

Since this is a console version of the game, you won't be able to draw the stickman and gallows,[4] so it will suffice to keep track of the number of guesses remaining before the game is lost. At minimum, your program must be able to perform the following functions:

[4] Well, I suppose you could draw it as "text art" or "ASCII art", and you're free to do so if you like, but you'll be making a graphical version of the game in the next problem anyway...

1. Select a random secret word that contains *at least four letters* from the provided lexicon.

2. Display the number of guesses remaining. The player starts the game with 8 guesses left.

3. Prompt the human player for successive letter guesses through the terminal until the game is over.

- If a guess is correct, reveal all occurrences of that letter in the appropriate positions in the secret word, keeping the other letters hidden as underscore (_) or dash (-) characters.
- If the secret word does not contain the letter guessed, add the letter to the list of incorrect guesses and deduct one from the guesses remaining.
- If the letter has been guessed before, inform the player and ask for a new guess.

4. Detect when the game is either won or lost, display a message accordingly, and reveal the secret word at the end.

This might also be a good time to remind you of the things you learned early on with Karel the Robot. You'll notice the form of this problem is very similar to what you already did in Assignment #2, except that we're no longer asking for specific intermediate steps. It's completely up to you to decide how you want to organize your program, but we suggest you try using top-down design and stepwise refinement to break it up into sub-problems. Identify the tasks that need to be completed and write functions for them if you can.

Inputs: The name of the lexicon file specified as a command line argument. For example, you may run your program as follows:

```
$ python3 my-p3.py cpsc231-lexicon.txt
```

Outputs: A guess-by-guess transcript of the game unfolding as you play Console Hangman with your computer program.

Example Problem 3 output:

```
Welcome to CPSC 231 Console Hangman!

The secret word looks like: _______
You have 8 guesses remaining.
What's your next guess? e
Nice guess!

The secret word looks like: ______e
You have 8 guesses remaining.
What's your next guess? a
Sorry, there is no "a".

The secret word looks like: ______e
Your bad guesses so far: a
You have 7 guesses remaining.
What's your next guess? o
Nice guess!

The secret word looks like: _o____e
Your bad guesses so far: a
You have 7 guesses remaining.
What's your next guess? t
Sorry, there is no "t".

The secret word looks like: _o____e
Your bad guesses so far: a t
You have 6 guesses remaining.
What's your next guess? u
Nice guess!

The secret word looks like: _o__u_e
Your bad guesses so far: a t
You have 6 guesses remaining.
What's your next guess? n
Nice guess!

The secret word looks like: _on_u_e
Your bad guesses so far: a t
You have 6 guesses remaining.
What's your next guess? f
Nice guess!

The secret word looks like: _onfu_e
Your bad guesses so far: a t
You have 6 guesses remaining.
What's your next guess? s
Nice guess!

The secret word looks like: _onfuse
Your bad guesses so far: a t
You have 6 guesses remaining.
What's your next guess? c
Nice guess!

Congratulations!
You guessed the secret word: confuse
```

Problem 4: Hangman Game

Now it's time to put the final touches on your Hangman game by adding a visual display of the stickman, the (partial) secret word, and the list of incorrect guesses. Create a *separate module* whose functions will take care of drawing the visual depiction of the aforementioned game elements to a graphics window on your screen. Do this by creating an additional Python (`.py`) file named in such a way that you can `import` and use it from a client such as your program from Problem 3. If you do this correctly, you can keep the code that handles the logic of the Hangman game separate from the code that creates the visual display, and you would only need to make very minimal changes to your previous program to add the new visuals.

At the end, your Hangman game should have a visual display that might look like what is shown in Figure 1. You're free to take artistic liberty when creating your visuals, as long as the following three elements are shown on the screen: the (partial) stickman and gallows, the (partially revealed) secret word, and the list of incorrect guesses. After the game is won or lost, provide an appropriate visual indicator (either graphical or text message) in your graphics window as well.

Draw the parts of the stickman and gallows in the order indicated on the right as incorrect guesses are accumulated. The complete "hanged" stickman may look like that shown in Figure 3 when all guesses have been exhausted. Some of the console output is now redundant with the visuals, but you may keep or omit it as you wish.

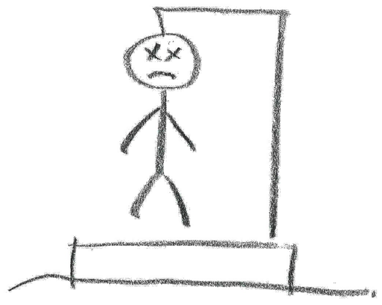

Figure 3: This is what the final "hanged" stickman may look like after 8 incorrect guesses. Draw the visual elements in the following order, one for each incorrect guess:

1. head
2. torso
3. left arm
4. right arm
5. left leg
6. right leg
7. gallows
8. face

Inputs: The name of the lexicon file specified as a command line argument. For example, you may run your program as follows:

```
$ python3 my-p4.py cpsc231-lexicon.txt
```

Outputs: A graphical display of the game unfolding as you play Hangman with your computer program.

Bonus Problem: Evil Hangman

If you play your Hangman game from Problem 4 a few times, you might start to find that it's pretty easy to beat, especially since the computer is restricted to choosing a random word from the 4000 most common words. Fortunately, it's easy for us to fix that—the same way many computer games "fix" their deficient AIs—by giving the computer player an unfair advantage. Your goal for this bonus problem is to create an evil, cheating Hangman game where the computer trounces the human player every time by not playing fairly!

You might have noticed that it is pretty easy to cheat at Hangman if you're playing the executioner. The game fundamentally relies on the player who chose the secret word to faithfully represent it when revealing correctly-guessed letters. However, if you chose the length of a word, but did not commit to a single word, or changed to a different word that fits the revealed pattern partway through the game, your opponent would have no way of knowing the better!

Here is what one cheating algorithm for the computer may work. After choosing a random word length, it finds and collects all the words of that length in the lexicon. When the human player guesses a letter, the computer will partition the words into "families" according to the patterns of letters that would be revealed for that guess.

For example, imagine a word length of four was chosen and the lexicon contained the following four-letter words: both, here, lead, make, many, time, well, and word. Suppose the human player guessed an "e". Then the word families according to their revealed letter patterns would look as shown on the right.

Word families for a guess of "e":

```
____ : both many word
_e_e : here
_e__ : lead well
___e : make time
```

Clearly, the computer must then choose one of these patterns to reveal. There are many ways to pick which word family to continue with, and each may have advantages and disadvantages. For now, we can go with the simple and effective strategy of choosing the family with the largest number of words. In this case, it's the `____` family, so the three words "both", "many", and "word" are kept, and the player is informed that he or she made an incorrect guess.

Next, suppose the player guesses an "o". The resulting families are shown on the right. Applying this strategy again, the word "many" is discarded, the pattern `_o__` is revealed, and play continues.

Word families for a subsequent guess of the letter "o":

```
_o__ : both word
____ : many
```

The game will end with one of the usual two outcomes of Hangman. First, the player may have been clever enough to pare your word list down to a single word and guess all the letters. In this case, you should congratulate the player profusely because it's an impressive feat considering all the scheming you've done! Second, the player may run out of guesses and be completely stumped, which, if you made the game correctly, will be by and large the most likely outcome. Then the computer can pick any random word remaining in the list, reveal it, and pretend that was the secret word it had chosen all along. The beauty of this setup is that the player would have no idea that the game were any different from that of Problem 4!

For this bonus problem, create and submit an Evil Hangman game that plays using the strategy described. Have the program choose a random word length between 4 and 9 letters, inclusive, to start. Then play the game out through an interface exactly has you had for Problem 4. Can you ever beat the game? Can you think of situations where the largest word family is not the best choice for the computer?[5]

Try to think of improvements to the "largest family" strategy that would make your computer even more evil. Perhaps you can program the computer to look ahead one or more turns, considering what the future outcome would be if the player were to guess different letters.[6] If you implement improvements over the basic strategy described here, submit with your program a brief description of your strategy and why it is better than the one described.

[5] For example, what happens when you have three words left in your list, "deal", "tear", and "skip", the player only has one guess remaining, and guesses an "e"?

[6] The idea of evaluating several moves ahead generalizes to what we call a *minimax* algorithm. Such algorithms have been shown to be very effective against humans in two-player games with finite possibilities for moves, such as checkers or chess. It can play a theoretically perfect game, provided the space of possible future moves can be searched completely. You can find the minimax strategy described in many texts on classic artificial intelligence.

Creative Bonus

Creativity plays a major role in many aspects of life, including your study of computer science! This bonus is here to encourage you to exercise your right-brain creativity in this course. Your TA will have liberty to award this bonus to the Hangman game that he or she deems the most creative submission. This could be judged in terms of aesthetics, design, quality of experience, entertainment value, or any combination of the above.

Basic stickmen, especially those drawn with computer-perfect lines, aren't particularly flattering. A natural approach to obtaining this bonus would be to improve the look of your game. You could also try animating the events of each turn or at the conclusion of the game. The Booksite Library even makes it easy for you to add sound effects! Do whatever you'd like to improve the gameplay experience, though if you deviate much from the "standard" Hangman of Problem 4, please submit a separate program for this bonus.

Java Bonus

Have we mentioned before that multi-lingual computer scientists are more employable? Even if you're not planning for a career in software development, solving problems in more than a single programming language helps you greatly to think about the problems at an algorithmic level, rather than merely at the implementation level. (That's a good thing.) As with the previous assignments, we will award you another bonus if you submit correct Java programs for Problems 3 and 4, in addition to your Python programs.[7]

[7] Note that you should consider your Python programs to be your de facto solutions for this assignment. You will not get credit for solving these problems unless your Python solutions are correct.

You may set up your Java programs however you'd like, as long as it is convenient enough for your TA to run them on the CPSC lab machines. There exists a version of our course text that uses the Java programming language, titled *Computer Science: An Interdisciplinary Approach*.[8] It has corresponding support libraries that can be found at `https://introcs.cs.princeton.edu/java/home/`. If you're not sure where to start with Java, you may find it most convenient to follow the instructions from that booksite. You may also use functionality from their Java Booksite Library if that helps you to create Java-language equivalents of your Python programs.

[8] Robert Sedgewick and Kevin Wayne. *Computer Science: An Interdisciplinary Approach*. Addison-Wesley, 2016

Submission

When you've completed the individual component, submit your two Python programs (`.py` files) for Problems 1 and 2. After completing the paired component, submit your solutions to Problems 3 and 4, including source files for additional modules you created, as `.py` files as well. Please indicate clearly, in the file names or otherwise, which Python files to run for each of the problems. If you completed the Java Bonus, submit the source of your all your Java classes (`.java` files) needed to run your programs for both problems.

Use the University of Calgary Desire2Learn system[9] to submit your assignment work online. Log in using your UofC eID and password, then find our course, CPSC 231 L01 and L02, in the list. Then navigate to Assessments → Dropbox Folders, and find the folder for Assignment #4 here. Upload your programs for Problems 1 and 2 in the individual folder, and your other program source files in the paired folder (if your partner hasn't already done so). One submission will suffice for each pair of students.

[9] `http://d2l.ucalgary.ca`

If you are using one or more of your grace "late days" for this assignment, indicate how many you used in the note accompanying your submission. Remember that late days will be counted against *both* partners in the paired component. If you completed any of the bonus problems, please indicate that here as well.

Credits

The idea of the Hangman game as an assignment was shamelessly borrowed from the CS 106A course taught at Stanford University. It was likely Eric Roberts who introduced this assignment there. The Evil Hangman bonus problem was adapted from that described by Keith Schwartz, also of Stanford University. The lexicon of most frequently used words in contemporary American English was obtained from `https://www.wordfrequency.info`.