The University of Texas
Computer Science

313E Programming Assignment 07
Evil Hangman

Setup

This assignment may be completed individually or with a pair programming partner. Be sure to include your and your partner's name and EID in the Python file header. If you are working alone, you may delete one of them.

Complete these steps to prepare for the Assignment.

Check	Description
▢	Download hangman.py, runner.py, dictionary.txt, and smallDictionary.txt. You will be working on the hangman.py file.
▢	Place all four files in the same folder/directory.
▢	You may not change the file names. Otherwise, the grading script will not work.

If you would like to turn on autosave in VSCode, click on **File -> Autosave.**

Problem Description

For this assignment, you will be programming Evil Hangman. In this version of the game, the program AI cheats, but in a very subtle and clever way. If you are not familiar with the game "Hangman", you can read the wikipedia page, and/or play a game online.

Evil Hangman is "**evil**" because the computer delays picking the secret word for as long as possible. The program starts with a list of words from the English dictionary. As the user guesses letters, the computer narrows down the list of "active" words, or words that could still be chosen as the secret word. The words in the active list all have the same pattern as the "word" that is displayed to the user based on the current guesses.

In other words, the program does not pick the secret word until it has to. Instead, it keeps a list of possible words based on user guesses.

A pattern consists of the letters that have been guessed correctly and the letters that have not been guessed yet. For example, if the current pattern is "-e-e-" then the words "beget", "beret", "bevel", "defer", "deter",

"level" and so forth would be in the list of active words. Those words "match" the pattern. The "-" characters are wildcard characters and represent any character that has not been guessed yet.

When the user guesses a letter, the evil AI will split the current list of active words into families, based on the possibilities for where the guessed letter could be placed.

Let us continue with the example pattern "-e-e-" and the current list of active words being ["beget", "beret", "bevel", "defer", "deter"]. If the user guesses the letter "t", there are 3 possibilities for the evil AI:

1. Make the pattern "-e-et"
2. Make the pattern "-ete-"
3. Keep the pattern "-e-e-" and count "t" as an incorrect guess

We can group the active words into separate families based on these 3 possibilities:

4. Make the pattern "-e-et": ["beget", "beret"]
5. Make the pattern "-ete-": ["deter"]
6. Keep the pattern "-e-e-" and count "t" as an incorrect guess: ["bevel", "defer", "level"]

Here is another explanation of how the program cheats. Note that the program doesn't actually pick a single word until it is forced to. Suppose that the user has asked for a 5-letter word. Instead of picking a specific 5-letter word, it considers all 5-letter words from the original dictionary as potential choices. Then as the user makes guesses, the program can't completely lie. It has to somehow fool the user into thinking that it isn't cheating. In other words, it has to cover its tracks. Your Hangman shall do this in a very specific way every time the make_guess method is called. Let's look at a small example.

Suppose that the dictionary contains just the following 9 words:

[ally, beta, cool, deal, else, flew, good, hope, ibex]

Now, suppose that the user guesses the letter 'e'. You now need to indicate which letters in the word you've "picked" are e's. Of course, you haven't picked a word, and so you have multiple options about where you reveal the e's. Every word in the set falls into one of five "word families:"

"- - - -": which is the pattern for [ally, cool, good]

"- e - -": which is the pattern for [beta, deal]

"- - e -": which is the pattern for [flew, ibex]

"e - - e": which is the pattern for [else]

"- - - e": which is the pattern for [hope]

Since the letters you reveal have to correspond to some word in your word list, you can choose to reveal any one of the above five families. There are many ways to pick which family to reveal – perhaps you want to steer your opponent toward a smaller family with more obscure words, or toward a larger family in the hopes of keeping your options open. In this assignment, in the interests of simplicity, we'll adopt the latter approach and always choose the largest of the remaining word families. In this case, it means that you should pick the family "- - - -". This reduces your set of words to:

[ally, cool, good]

Since you didn't reveal any letters, count this as a wrong guess.

Let's see a few more examples of this strategy. Given this three-word set, if the user guesses the letter 'o;, then you would break your word list down into two families:

“- o o -”: containing [cool, good]

“- - - -”: containing [ally]

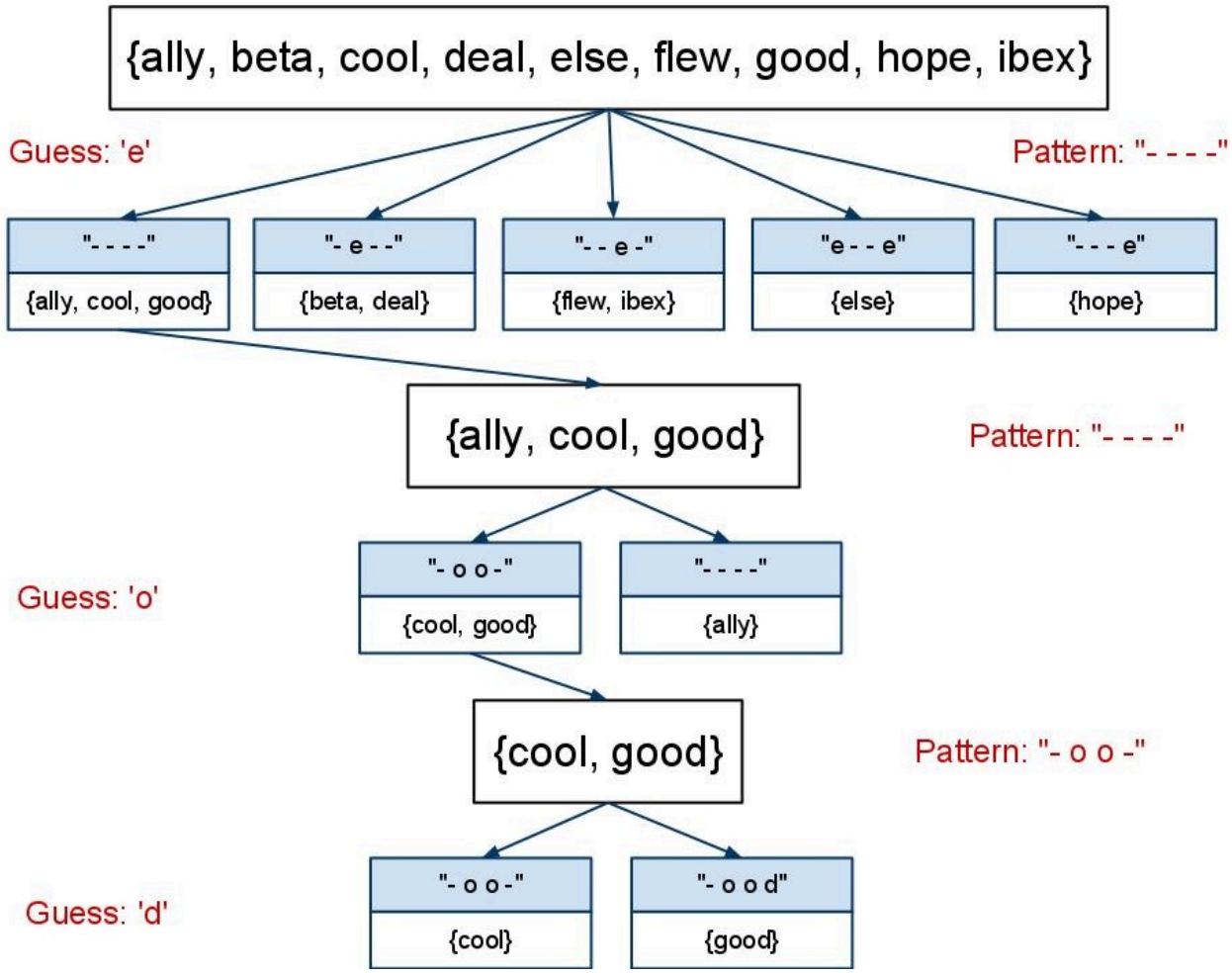
The first of these families is larger than the second, and so you choose it, revealing two 'o's in the word and reducing your set of words to

[cool, good]

In this case the user has correctly guessed a letter because there are two occurrences of 'o' in the new pattern.

But what happens if your opponent guesses a letter that doesn't appear anywhere in your word list? For example, what happens if your opponent now guesses 't'? This isn't a problem. If you try splitting these words apart into word families, you'll find that there's only one family – the family “- o o -” in which t appears nowhere and which contains both “cool” and “good”. Since there is only one word family here, it's trivially the largest family, and by picking it you have to maintain the word list you already had and you would count this as an incorrect answer.

Here is a useful diagram that shows the winnowing process based on user guesses.



And here is a link to a more detailed, explanation for how to break the current word list up into families based on the current pattern and current guess.

The evil AI must choose which family to go with based on the difficulty level. A description of how to select the family is provided in the method description for sort.

Requirements

Our version of the game **will NOT** have any graphics or drawings of the gallows. It will be played through the command line (like the wordle project).

We will be providing you the driver program that handles the input and output, **runner.py**. You will be writing a class, **Hangman**, that manages the details of Evil Hangman. This class keeps tracks of the possible words from a dictionary during rounds of Evil Hangman, based on the guesses so far. You will be working in the **hangman.py** file. The methods that you must implement in your program are described below.

Type	Description
Constructor (__init__)	Create a new Hangman from the provided set of words and phrases. You may initialize attributes here as needed. Pre-condition: words != null, words.size() > 0 :param **words**: A set with the words for this instance of Hangman. :param **debug**: Set to True if we should print out debugging to command line
num_words	Get the number of words in this Hangman of the given length. :param **length**: The given length to check. :**return**: The number of words in the original Dictionary with the given length.
prep_for_round	Get for a new round of Hangman. Think of a round as a complete game of Hangman. You should update variables here as needed. Pre-condition: word_len > 0, num_guesses >= 1 :param **word_len**: The length of the word to pick this time. :param **num_guesses**: The number of wrong guesses before the player loses the round. :param **diff**: The difficulty for this round.
num_words_current	The number of words still possible (active) based on the guesses so far. :**return**: The number of words that are still possibilities based on the original dictionary and the guesses so far.
get_guesses_left	Get the number of wrong guesses the user has left in this round (game) of Hangman. :**return**: The number of wrong guesses the user has left in this round (game) of Hangman.
get_guesses_made	Return a String that contains the letters the user has guessed so far during this round. :**return**: A String that contains the letters the user has guessed so far during this round.

get_pattern	Get the current pattern. The pattern contains '-''s for unrevealed (or guessed) characters and the actual character for "correctly guessed" characters. :**return**: the current pattern.
make_guess	Update the game status (pattern, wrong guesses, word list), based on the given guess. :param **guess**: the current guessed character :**return**: A dictionary with the resulting patterns and the number of words in each of the new patterns. The return value is for testing and debugging purposes.
get_map_pattern	Returns a dictionary that maps patterns to a list of words that follow said pattern. Pre-condition: Guess has not been guessed before in this round. :param **guess**: The current guessed character. :**return**: A dictionary that maps patterns to a list of words that follow said pattern.
make_pattern	Builds possible word patterns for each word based on the user's guess and the previous pattern. Pre-condition: guess has not been guessed before in this round, word is not None. :param **word**: The word to build the pattern for. :param **guess**: The current guessed character. :**return**: The pattern for the given word based on the user's guess and the previous pattern.
sort	Return sorted list of entries. First sort by number of words in word list, then the number of dashes in the pattern, next to the pattern, and finally the word list itself Pre-condition: entries must be a list of tuples with four elements each (length of the word list, dash count, pattern, word list). You must implement merge sort. :param **entries**: The Family tuples to sort. :returns: a new sorted list.
order_entries	For each key-value pair of (pattern, word list) in word_family, a Family tuple with four elements each (length of the word list, dash count, pattern, word list) is created and added to a list. The list of Family tuples is then sorted (hint: you can use another function you have written!). Pre-condition: word_family is not None. :param **word_family**: A dictionary containing patterns as keys and lists of words as values. :**return**: A sorted list of Family tuples (length of the word list, dash count, pattern, word list).
get_secret_word	Return the secret word this Hangman finally ended up picking for this round. Return the secret word this Hangman finally ended up picking for this round. If there are multiple possible words left, one is selected at random. The seed should be initialized to 0 before picking. :**return**: return the secret word the manager picked.

Ties resulting from tuples can be handled using tuple element comparison. Tuples can be compared using comparison operators such as **==**, **<**, **<=**, **>**, **>=**, and **!=**. When comparing tuples with more than two elements, if

the first elements do not satisfy the comparison, the comparison continues with the second elements, and so on. For instance, given the tuples **("a", "b", "c")** and **("a", "e", "f")**, if we want to compare **("a","b", "c") > ("a", "e", "f")** and since "**a" == "a"**, the comparison will proceed to compare the next elements, "**b"** and "**e"**.

For each call to make_guess, find all of the word families based on the current list of active words (not the original dictionary, except for the first guess) and pick the one that is the hardest.

The hardest word list / family is defined as the one with the largest number of words with certain tiebreakers. (This family becomes the new list of words for the next round of the game.) If there is a tie for the family with the most words (two of the word families are of equal size), pick the family that reveals the fewest total characters. If there is still a tie, pick the family that has the pattern that is "smallest" based on the ordering of the patterns. Note that this is done for you if you just use the comparison operators between strings.

Summary of hardest word list / family:

1. Family with the maximum elements
2. If there are two or more families with the maximum number of elements break the tie by picking the one that reveals the fewest characters
3. If there are two or more families with the maximum number of elements and reveal the same number of characters break the tie by picking the pattern that is "smallest" based on the ordering of strings (the ordering of the strings is based on their Unicode values, but remember that the comparison is already implemented for you using ==, <, etc.).

This should be implemented in your sort() and order_entries() methods.

Initially ignore the user's choice for difficulty and implement the algorithm above. When you have that working then alter the make_guess method to take into account the difficulty. This difficulty chooser has been implemented for you in get_diff() and get_difficulty(). Use them to implement the following logic. It should be as simple as just using get_difficulty() as the index to pick the list / family.

If the difficulty is HARD then the program picks the word list as described above. HARD always picks the hardest word list / family.

If the difficulty is MEDIUM the program picks the hardest word list / family 3 times in a row, then the second hardest word list, then the hardest word list / family 3 times in a row, then the second hardest word list / family. This pattern repeats until the game is over. The game should be a little less difficult if the program doesn't always pick the hardest word list / family. If their is only one word list / family and it is time to pick the second hardest, just pick the single word list / family. Then go back to picking the hardest. In other words, the behavior for picking word lists at this difficulty level is

Medium behavior for picking a new list of words in makeGuess method:
Guess 1, PICK HARDEST WORD LIST
Guess 2, PICK HARDEST WORD LIST
Guess 3, PICK HARDEST WORD LIST
Guess 4, PICK SECOND HARDEST WORD LIST IF IT EXISTS, OTHERWISE PICK HARDEST
Guess 5, PICK HARDEST WORD LIST
Guess 6, PICK HARDEST WORD LIST
Guess 7, PICK HARDEST WORD LIST
Guess 8, PICK SECOND HARDEST WORD LIST IF IT EXISTS, OTHERWISE PICK HARDEST
and so forth ...

If the difficulty is HangmanDifficulty.EASY the program alternates between picking the hardest word list and the second hardest word list if it exists:

Easy behavior for picking new list of words in makeGuess method:
Guess 1, PICK HARDEST WORD LIST
Guess 2, PICK SECOND HARDEST WORD LIST IF IT EXISTS, OTHERWISE PICK HARDEST
Guess 3, PICK HARDEST WORD LIST
Guess 4, PICK SECOND HARDEST WORD LIST IF IT EXISTS, OTHERWISE PICK HARDEST
Guess 5, PICK HARDEST WORD LIST
Guess 6, PICK SECOND HARDEST WORD LIST IF IT EXISTS, OTHERWISE PICK HARDEST
Guess 7, PICK HARDEST WORD LIST
Guess 8, PICK SECOND HARDEST WORD LIST IF IT EXISTS, OTHERWISE PICK HARDEST
and so forth ...

If it is time to pick the second hardest word list, but there is only one-word list, then that single word list is the pick. In this case, the second hardest does not "roll over" to the next round.

This assignment includes the following technical requirements:

1. You may **not** make any changes to runner.py. You will only be editing **hangman.py**.
2. **You may not use any built-in sorting functions or methods.** You **must** implement **merge sort or a stable version of quick sort** for the sort() method. You may **not** use selection sort, insertion sort, or any other sorting method.
3. Before turning in your assignment, you **must** set **DICTIONARY_FILE = "dictionary.txt".**
4. **You may not change the names or parameters of the functions listed. They must have the functionality as given in the specifications. You can always add more functions than those listed.**
5. **You may not import any additional external libraries in your solution besides random.**

Input

You can run the program by typing the following command into your terminal:

```
python3 runner.py
```

We have provided you two files with lists of words for this assignment. For testing purposes, you may use the smaller sample dictionary found in the file **smallDictionary.txt**. To change which file you will be using when running your program, you should uncomment the respective dictionary at the top of **hangman.py**. You can assume that all input files are structured correctly.

Additionally, we have provided “debugging” text that you can enable or disable located in hangman.py. If you would like to enable debugging text, set **DEBUG = True.** If you would like to disable debugging text, set **DEBUG = False**.

Output

Code to produce the output already exists in the file **runner.py**. The existing code will call the methods that you wrote. For all example outputs, we will be using the file smallDictionary.txt as the list of words.

However, before turning in your assignment to Gradescope, you **must** set **DICTIONARY_FILE = "dictionary.txt" and** set **DEBUG** to **False.**

Below is a sample output of a successful run.

```
Welcome to the CS313E hangman game.

What length word do you want to use? 4
How many wrong answers allowed? 4
What difficulty level do you want?
Enter a number between 1 (EASIEST) and 3 (HARDEST): 1
Guesses left: 4
Guessed so far: []
Current word: ----

Your guess? a
Yes, there is one a
Guesses left: 4
Guessed so far: ['a']
Current word: --a-

Your guess? n
Sorry, there are no n's
Guesses left: 3
Guessed so far: ['a', 'n']
Current word: --a-

Your guess? e
Yes, there is one e
Guesses left: 3
Guessed so far: ['a', 'e', 'n']
Current word: -ea-

Your guess? s
Sorry, there are no s's
Guesses left: 2
Guessed so far: ['a', 'e', 'n', 's']
Current word: -ea-

Your guess? t
Sorry, there are no t's
Guesses left: 1
Guessed so far: ['a', 'e', 'n', 's', 't']
Current word: -ea-

Your guess? d
Yes, there is one d
Guesses left: 1
Guessed so far: ['a', 'd', 'e', 'n', 's', 't']
Current word: dea-
```

```
Your guess? l
Yes, there is one l
Answer = deal
You beat me

Another game? Enter y for another game, anything else to quit:
```

Below is a sample output of a run where both a guess is not a string, and a repeat guess is given:

```
Welcome to the CS313E hangman game.

What length word do you want to use? 4
How many wrong answers allowed? 4
What difficulty level do you want?
Enter a number between 1 (EASIEST) and 3 (HARDEST): 1
Guesses left: 4
Guessed so far: []
Current word: ----

Your guess? e
Yes, there is one e
Guesses left: 4
Guessed so far: ['e']
Current word: ---e

Your guess? 5
That is not an English letter.
Your guess? a
Sorry, there are no a's
Guesses left: 3
Guessed so far: ['a', 'e']
Current word: ---e

Your guess? a
You already guessed that! Pick a new letter please.
Your guess? o
Yes, there is one o
Guesses left: 3
Guessed so far: ['a', 'e', 'o']
Current word: -o-e

Your guess? h
Yes, there is one h
Guesses left: 3
Guessed so far: ['a', 'e', 'h', 'o']
Current word: ho-e

Your guess? p
Yes, there is one p
Answer = hope
```

```
You beat me

Another game? Enter y for another game, anything else to quit:
```

Below is a sample output of a run where numbers are not correctly given for setup:

```
Welcome to the CS313E hangman game.

What length word do you want to use? a
Error: Please enter a valid integer for word length.
What length word do you want to use? 4
How many wrong answers allowed? a
Error: Please enter a valid integer for number of wrong guesses.
How many wrong answers allowed? 4
What difficulty level do you want?
Enter a number between 1 (EASIEST) and 3 (HARDEST): a
Error: Please enter a valid integer for difficulty level.
What difficulty level do you want?
Enter a number between 1 (EASIEST) and 3 (HARDEST): 1
Guesses left: 4
Guessed so far: []
Current word: ----
```

Below is a sample output of a run where debugging is enabled:

```
Welcome to the CS313E hangman game.

Number of words of each length in the dictionary:
2: 0
3: 0
4: 9
5: 0
...
23: 0
24: 0
What length word do you want to use? 4
How many wrong answers allowed? 4
What difficulty level do you want?
Enter a number between 1 (EASIEST) and 3 (HARDEST): 3
Guesses left: 4
DEBUGGING: Words left: 9
Guessed so far: []
Current word: ----

Your guess? e
DEBUGGING: Picking hardest list.
DEBUGGING: New pattern is: e--e. New family has 1 words.
DEBUGGING: Based on guess here are the resulting patterns and
number of words in each pattern:
Pattern: ----, Number of words: 3
Pattern: --e-, Number of words: 2
Pattern: e--e, Number of words: 1
```

```
Pattern: -e--, Number of words: 2
Pattern: ---e, Number of words: 1
END DEBUGGING

Yes, there are 2 e's
Guesses left: 4
DEBUGGING: Words left: 1
Guessed so far: ['e']
Current word: e--e

Your guess? a
DEBUGGING: Picking hardest list.
DEBUGGING: New pattern is: e--e. New family has 1 words.
DEBUGGING: Based on guess here are the resulting patterns and
number of words in each pattern:
Pattern: e--e, Number of words: 1
END DEBUGGING

Sorry, there are no a's
Guesses left: 3
DEBUGGING: Words left: 1
Guessed so far: ['a', 'e']
Current word: e--e

Your guess? y
DEBUGGING: Picking hardest list.
DEBUGGING: New pattern is: e--e. New family has 1 words.
DEBUGGING: Based on guess here are the resulting patterns and
number of words in each pattern:
Pattern: e--e, Number of words: 1
END DEBUGGING

Sorry, there are no y's
Guesses left: 2
DEBUGGING: Words left: 1
Guessed so far: ['a', 'e', 'y']
Current word: e--e

Your guess? l
DEBUGGING: Picking hardest list.
DEBUGGING: New pattern is: el-e. New family has 1 words.
DEBUGGING: Based on guess here are the resulting patterns and
number of words in each pattern:
Pattern: el-e, Number of words: 1
END DEBUGGING

Yes, there is one l
Guesses left: 2
DEBUGGING: Words left: 1
Guessed so far: ['a', 'e', 'l', 'y']
```

```
Current word: el-e

Your guess? s
DEBUGGING: Picking hardest list.
DEBUGGING: New pattern is: else. New family has 1 words.
DEBUGGING: Based on guess here are the resulting patterns and
number of words in each pattern:
Pattern: else, Number of words: 1
END DEBUGGING

Yes, there is one s
Answer = else
You beat me

Another game? Enter y for another game, anything else to quit:
```

Grading

Visible Test Cases - 75/100 points

The program output exactly matches the sample solution's output. To receive full credit, the program must produce the correct output based on all requirements in this document and pass all the test cases.

Hidden Test Cases - 15/100 points

The program output exactly matches the sample solution's output. To receive full credit, the program must produce the correct output based on all requirements in this document and pass all the test cases.

Style Grading - 10/100 points

The hangman.py file must comply with the PEP 8 Style Guide.

If you are confused about how to comply with the style guide, paste the error or the error ID (e.g. C0116 for missing function or method docstring) in the search bar here in the Pylint Documentation, and you should find

The TAs will complete a manual code review for each assignment to confirm that you have followed the requirements and directions on this document. Deductions will occur on each test case that fails to follow requirements.

Submission

Follow these steps for submission.

Check	Description
▢	Verify that you have no debugging statements left in your code.

Check	Description
▢	Submit **only** hangman.py (the starter code file with your updates) to the given assignment in Gradescope. This will cause the grading scripts to run.
▢	If you have a partner, make sure you are submitting it for you **and** your partner on Gradescope. See this Gradescope documentation for more details.
▢	When the grading scripts are complete, check the results. If there are errors, evaluate the script feedback, work on the fix, test the fix, and then re-submit the file to Gradescope. You can submit as many times as necessary before the due date. NOTE: By default, only the most recent submission will be considered for grading. If you want to use a previous submission for your final grade, you must activate it from your submission history before the due date.

Academic Integrity

Please review the Academic Integrity section of the syllabus. We will be using plagiarism checkers, and we will cross-reference your solution with AI-generated solutions to check for similarities. Remember the goal in this class is not to write the perfect solution; we already have many of those! The goal is to learn how to problem solve, so:

- Don't hesitate to ask for guidance from the instructional staff.
- Discuss with peers about **only** bugs, their potential fixes, **high-level** design decisions, and project clarifying questions. Do not look at, verbalize, or copy another student's code when doing so.

Attribution

Thanks to Dr. Carol Ramsey and Professor Mike Scott for the instructions template and this assignment.