This homework will require you to create functions that output text files in addition to variable outputs. Text files cannot be checked against the solution output using `isequal()`, but there is another function you can use to compare text files called `visdiff()`.

Suppose your output file is called `'textFile1.txt'` and the solution function produces the file `'textFile1_soln.txt'`. From the Command Window, type and run the following command:

```
visdiff('textFile1.txt','textFile1_soln.txt');
```

At this point, a new window will pop up. This is the MATLAB File Comparison Tool. It will not only tell you if the selected files match, but it will also tell you exactly what and where all of the differences are. Use this tool to your advantage. **Please note that sometimes the comparison will say, "No differences to display. The files are not identical, but the only differences are in end-of-line characters." Do not be alarmed if you see this; you will still receive full credit.**

Please keep in mind that your files must be named exactly as specified in the problem descriptions. The solutions will output files with '`_soln`' appended before the extension. Your output filename should be identical to the solution output filename, excluding `'_soln'`. Misspelled filenames will result in a score of 0. You will still need to use `isequal()` to compare non-text-file function outputs.

Also note, you can start the File Comparison Tool by clicking on "Compare" in the Editor ribbon if that is easier for you.

MATLAB has lots of additional functions for reading/manipulating low-level text files, but because we want you to learn the fundamentals of low-level file I/O, we have banned `fileread()` and `textscan()` **for all problems** on this homework. Use of either of those functions on any problem will result in a 0 for that problem.

Happy coding!
~Homework Team

Function Name: `banFinder`

Inputs:

1. *(char)* The name of a text file containing MATLAB code
2. *(char)* The banned function

Outputs:

(none)

File Outputs:

1. The result text file with `'Result.txt'` appended at the end of its name

Function Description:

Write a banned function finder called `banFinder()` that takes in the name of a text file containing some MATLAB code and the name of a banned function. `banFinder()` should then search through the text file and determine if the banned function was used, determine how many times it was used (if at all), and output a text file that gives the result. Ensure that you look for the banned function in lines of code, not in commented lines (because commented lines do not actually run). You do not have to account for the banned function occurring as a substring of a variable name.

Depending on whether the banned function is found or not, you should output a result text file in one of two formats. The output text file name should be the same as the input function text file name, but with `'Result.txt'` appended at the end. For example, `'colorByNum.txt'` should become `'colorByNumResult.txt'`.

Example:

`banFinder()` is run with the following inputs: `'colorByNum.txt','num2str'.`
If the `num2str` function was used twice, the output text file `'colorByNumResult.txt'` should read:

```
The num2str function was used 2 time(s).
Fail.
```

If `num2str` function was not used, the output text file `'colorByNumResult.txt'` should read:

```
The num2str function was not used.
Pass.
```

Notes:

- You do not need to check the first line since that is guaranteed to be the function header.
- Comment lines will always start with a `'%'` as the first character.
- Lines will either have code or comments, but will never have both.
- Banned functions will never appear as a part of a variable name or string.

Function Name: `roommateMatch`

Inputs:

1. *(char)* The name of a text file containing the first roommate's survey answers
2. *(char)* The name of a text file containing the second roommate's survey answers

Outputs:

1. *(char)* A sentence describing the compatibility between two roommates

Function Description:

A significant part of the college experience may be living away from home for the very first time. Finally your own space… to be potentially shared with a complete stranger. To ensure that your college roommate is top notch, you will write a function that compares two roommate surveys and outputs a sentence describing the compatibility of the match. The output will be formatted as:

```
'<1st roommate name> and <2nd roommate name> have a <percent>% roommate compatibility score.'
```

Both files will be formatted the following way.

```
1     Name: Melanie March
2     Earliest Class: 10:00 AM
3     How late do you stay up:
4        Early
5     X  Late
6        Really Late
7     Food Preferences: Vegetarian
8     Tell us about yourself:
9     i am 20 years old and am bringing my cat, fluffy, with me :)  i also
10    really really love to sing in the shower.  I hope my roommate is
11    musical like mee ;D
12
```

The first 8 lines will always define the same information (Name on first line, earliest class on second, etc). Lines 4-6 will have the strings `'Early'`, `'Late'`, and `'Really Late'`, respectively and there will always be one and only one option selected, denoted by an `'X'` on the appropriate line. The roommates will always write something on line 9 and may continue writing for an unknown number of lines.

The criteria for deciding how compatible two roommates are includes comparing the time of their first class, how late each stays up, and how talkative they are. Each one of those 3 criteria is worth 20 points for a maximum of 60 points. The points for each criteria should be calculated as follows.

First Class Compatibility:

Divide the difference in hours between the roommates' first classes by 12. Then subtract this number from 1, multiply by 20 and round to the nearest integer. For example, roommates with their first classes starting at 11:00 AM and 1:00 PM will receive a score of 17 points because the difference in those times is 2 hours and `round((1 - 2 / 12) * 20)` ⇒ `17`

You have been given a helper function `timeDiff()` that will calculate time differences for you. Type `help timeDiff` in the Command Window for information on how to use the helper function.

"Stay Up Late" Compatibility:

Award 20 points to roommates who stay up to the same time, 10 points to those who stay up to adjacent levels of lateness ("Early" and "Late", or "Late" and "Really Late"), and 0 points to those who stay up "Early" and "Very Late"

Talkativity Compatibility:

This will be based on the total number of characters in each roommates' response is to the last prompt (Tell us about yourself). You should concatenate each roommates' entire response into a single string (not including newline characters or leading/trailing whitespace). The shorter response's total number of characters will be divided by the longer response's total number of characters. That quotient will be multiplied by 20 and rounded to the nearest integer to get the total score for this section.

Once all of these scores have been calculated, the total percentage of compatibility will be the sum of all of the subscores divided by 60, expressed as a percentage between 0 and 100 rounded to the nearest percent.. Output this value in the string output as shown in the beginning of the description.

Notes:

- You can use `'%%'` in `sprintf()` to put a `'%'` in a string.
- You can use `strtrim()` to remove leading and trailing whitespace from a string.

Function Name: `leaderboard`

Inputs:

1. *(char)* The name of a text file containing a game leaderboard
2. *(char)* A gamer tag
3. *(double)* A final score

Outputs:

1. *(char)* A string describing your standing

File Outputs:

1. An updated leaderboard

Function Description:

In case you haven't noticed, you are going to have a working snake game by the end of the semester and any respectable game needs a leaderboard! This function will take in the filename of a leaderboard, a gamer tag and the final state of that gamer's board. The function will then update the leaderboard and output the gamer's rank. The updated leaderboard should have the same name as the input, but with `'_updated'` appended to the filename.

The leaderboard file will always be organized as:

```
<gamerTag1>,<score1>
<gamerTag2>,<score2>
...
<gamerTagN>,<scoreN>
```

A player's score for snake is just the length of the snake at the end of the game. The file will not necessarily have the gamers sorted by score. The updated leaderboard file will be the same as the input file except that the new player and score will be added to the end of the file. However, if the player was already listed in the file and they scored higher this time than they did previously, you should update that entry in the new file and not append a new line. If the player already exists in the file and they did worse than their high score, the leaderboard should be unchanged.

The text output for this function is of the form:

```
<prefix string> <tied string> OR <rank string>
```

`<prefix string>` is `'You did worse than last time!'` if the player scored lower than their existing highscore. It is `'You beat your own highscore!'` if the player beat their own highscore. In any other case, `<prefix string>` is empty.

`<tied string>` is `'You are tied with <num> other player(s) for a rank of <rank>.'` if there are multiple players with the same score. `<num>` is the number of other

players (not including yourself) that are tied and `<rank>` is the index of the first occurrence of the player score in a sorted vector of all the scores.

`<rank string>` is `'Your rank is <rank>.'` where rank is defined the same as above. This case is used when the input player is the only person with this score.

Function Name: `chooseAdventure`

Inputs:

1. *(char)* The name of a text file with the start of your story

Outputs:

(none)

File Outputs:

1. A text file of the completed story

Function Description:

Have you ever spent time reading through the countless possible endings in a Choose Your Own Adventure book? You'll never have to do that with this problem - there's only one possible outcome for each story! The story starts at the beginning of the given text file, and continues until it reaches a tag in the format of:

`'<filename|line number|character number>'`

Such a tag indicates a jump to a different part of a different text file. For example, if the tag is `'<storyPartTwo.txt|5|8>'`, the story includes up to the character before the `'<'` and continues in the file `'storyPartTwo.txt'`on character 8 of line 5. Tags **may be** split across multiple lines of a the text file, such as:

```
Once upon a time, <storyPartThree.txt|
20|13> I learned how to use MATLAB
```

The completed story does not include any of the characters from the tag, including the angle brackets (`'<'`, `'>'`). When you start reading from a new file, you should start writing on a new line. So in the above example, if `storyPartThree.txt` contained the text `'there was a young prince.'` then the final file would read:

```
Once upon a time,
There was a young prince.
```

The story ends when it reaches the end of a text file.

The completed story must be saved in a file with the same name as the original in the function input with `'_chAdv'` appended before the `'.txt'`. For example, if the original input is called `'story1.txt'`, the output file would be named `'story1_chAdv.txt'`.

Notes:

- Line 1 means start on the first line of the text file, character 1 means start on the first character of the line.
- `'<'` and `'>'` will only be used for jumps to new story text files.
- You will never jump to a location between or on top of angle brackets.

- A story can jump to a new location in the same text file, and a story can jump any number of times.
- A story will not jump to a line that does not exist in a new file or to a character number that exceeds the number of characters in the line.
- Be careful with when you open/close files.
- You SHOULD include a single newline character at the end of your file. This is to match the solution file which prints a newline, not because it is specified in the problem description.

Function Name: `reviewReviewer`

Inputs:

1. *(char)* The name of a text file that contains a product review
2. *(char)* A string of space-delimited positive keywords
3. *(char)* A string of space-delimited negative keywords

Outputs:

1. *(char)* A string indicating how many stars the reviewer gave the product

Function Description:

It's the 21st century and we *still* have to give a number of stars when posting a review of a product online. Ridiculous! Why can't the stars be calculated based on the content of our reviews? This is where you come in -- you're going to write a function to put an end to those extra clicks and those hours of pondering just how many stars a product really deserves.

You have three inputs, the first is the filename of the text file that contains a review of a product. You have to count all occurrences of positive keywords (given in the second input) and negative keywords (given in the third input). The review is deemed 5 stars when there are more than twice as many positive words than negative words, 4 stars if the positive words outnumber the negative words, 3 stars if there are as many positive words as there are negative ones, 2 stars if there are less positive words than negative words, and 1 star if there are more than twice as many negative words than positive words. Before checking the file for positive and negative words, you should remove all punctuation. For this problem, punctuation is defined as any of the following characters: `, . : ' " ; ? !` (spaces are NOT considered punctuation and should not be removed).

The first line of the text file will be in the format `'<Product Name> Review'`. You must give your output in the form:

`'The reviewer gave a <# of stars> star review of the <Product Name>.'`

Notes:

- You should not count words that appear as substrings of longer words. For example, if one of the positive words is `'joy'`, your function should not count `'joystick'` as an occurrence of this word.
- Your search should be case-insensitive, although there is no guarantee about the case of the inputs or the review text file.

Hints:

- Remember `gibberish()`...

EXTRA CREDIT

Function Name: `fileShift`

Inputs:

1. *(char)* The name of a text file
2. *(double)* An amount to shift the text

Outputs:

(none)

File Outputs:

1. A text file of the shifted text

Function Description:

If you thought `puzzleBox` was fun, then buckle up! Instead of shifting character arrays around, you will now be shifting text files! All of the text in the file should be shifted by the given shift and text from the end of the file should wrap around to the beginning of the file. The shifted file should be saved as

`<filename>_shifted.txt`

Where `<filename>` is replaced by the name of the input file excluding the file extension. For example, if the input file was named `shakespeare.txt` and contained the following text:

```
To be, or not to be
That is the question
```

And a shift of 5 was given as the second input, then the output file would be saved as `shakespeare_shifted.txt` and would contain the following text:

```
stionTo be, or not
to beThat is the que
```

Note that the number of characters on each line remains the same. Characters from the end of line n get shifted onto line n + 1 and characters from the end of the file wrap around to the beginning of the file. Your function should work for any shift value, both positive or negative. In the case of a negative shift, characters from the beginning of the file get shifted from the beginning of the file to the end.

Notes:

- Remember that strings are just vectors of chars.
- There will not be any empty lines in the files.

Hints:

- Be mindful of newline characters.