Notice

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 or right clicking on one of the files you want to compare and selecting "Compare Against" 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.

Finally, please remember to **close** all files that you open. The use of `fclose('all'), fclose all,` or `close all` is not permitted, so make sure to close each file individually. Failure to close files appropriately may result in loss of points.

Happy coding!
~Homework Team

Function Name: `nationalTreasure`

Inputs:

1. (*char*) The name of the file containing which letters to extract
2. (*char*) The name of the file to pull letters from

Outputs:

1. (*char*) A string containing the deciphered code

Function Description:

Back in 2004, the possibly greatest actor of our time, Nicolas Cage, starred in possibly the greatest movie of our time, *National Treasure*. In the movie, Cage finds an ottendorf cipher on the back of the Declaration of Independence as part of a series of clues to find the treasure and employs a child to run and look up all the letters that he needs, which is horribly inefficient. If he had taken CS1371 at Georgia Tech, he could have just written a program to do it for him in MATLAB!

Your job is to write a function called `nationalTreasure()` that takes in two files. The first is a text file that contains the "coordinates" for the cipher. Each line will correspond to a specific letter, and will be formatted as follows:

```
<line number>-<word number>-<letter number>
```

For example, the line

```
'5-3-1'
```

Corresponds to the 5th line, 3rd word, and first letter of that word. Additionally, instead of the above format, a line could just be the string

```
'space'
```

which means to insert a space. You are to use the information in the first file to extract a message from the second file, which will just be a text file to pull letters from. The output is a string containing the letters and spaces pulled from the second file in the order that they are in the cipher file.

Notes:

- The letter "coordinates" are guaranteed to be valid (i.e. we won't ask for the 5th letter of a word that has only 4).
- Words are separated by spaces only. Things like `'yes,so'` or `'ice-cream'` are considered a single word.
- Remember to close all your files!

Hints:

- Use `fclose()` and then `fopen()` to reopen a file and start reading from the top again.

Function Name: `tankTrials`

Inputs:

1. *(char)* The name of a text file containing the data for old tanks
2. *(char)* The name of a text file containing the data for new tanks

Outputs:

none

File Outputs:

1. A text file containing details of the tests

Function Description:

During your previous armored campaign, you discovered that your arsenal of powerful beasts of steel were being outmatched by your devious opponents. Thankfully, your team of research engineers were working hard to come up with new tank models to defeat the current era of armor. Since your research engineers are only good at coming up with new tank prototypes, you have decided to use MATLAB to accelerate testing and determine the viable designs.

Given a text file of old tanks and another text file of new tank designs, determine how many old tanks your new tank designs can defeat, and the highest ranking defeated old tank.

The list of current(old) tanks and lists of prototype(new) tanks (`'testVehicles#.txt'`) are formatted as denoted by the headers at the top of each file. The list of current tanks are formatted:

```
<Tank name>, <Plate thickness>, <Angle>
```

The lists of prototype tanks are formatted:

```
<Tank name>, <Firepower>
```

A new tank will defeat an existing(old) tank if its firepower is greater than or equal to the effective armor thickness (equation in notes) of the existing tank. You will need to keep track of how many existing vehicles the prototype tank can defeat.

You will also need to keep track of which existing tank has the strongest *defeatable* armor for each tank prototype. For example, if the prototype tank has 260 firepower and the existing tanks have 180, 255 and 270 effective armor, you will output the name of the tank with 255 effective armor because the prototype tank cannot defeat the armor of the tank with 270 effective armor.

The output file name will be `'<input filename>_results.txt'`. The first line of your output file should read:

```
Tank Trial Results:
```

Your output file should have one line for each prototype tank. Each line should be formatted as follows:

```
The <prototype tank> was able to defeat <number of existing vehicles> old
vehicles! The toughest opponent it beat was the <strongest defeated tank>.
```

If the prototype tank was not able to beat any existing vehicles, output the line:

```
'The <prototype tank> was not able to defeat any old vehicles.'
```

Notes:

- Effective armor value can be calculated using: $Effective\ Armor = \frac{Plate\ Thickness}{cos(angle)}$
- There should be no newline character at the end of your file. Check the solution file for the correct output formatting if you are unsure.
- The first line in the input files are the headings for the columns.
- The output text file should be printed in the order that the tanks appear in in the prototype designs file.

Hints:

- The `cosd()` function will allow you to use values in degrees without converting to radians.
- You can close and reopen a file to read it again from the beginning.

Function Name: `arr2text`

Inputs:

1. (*double*) A non-empty array of positive integers
2. (*char*) A filename

Outputs:

none

File Outputs:

1. A formatted table containing the array values

Function Description:

Sometimes it is important to be able to share formatted data but you don't have the luxury of being able to send an Excel sheet, Word document or any other type of rich-text document. In these cases, a plain-ol' text file will have to suffice. However, your tables can still look good! In this problem, you will be given an array of data that you need to convert into a formatted table and write that table to a text file. Here's the specifications for the formatting:

- Each column of the table should be the width of the largest number in that column.
- Each row should be delimited by `-`'s
- Each column should be delimited by `|`'s
- The intersection of the row and column delimiters should be +'s
- All numbers should be left-aligned in their cell and padded with spaces

The formatted array should be written to a file specified by the second input.

Here is an example:

```
array = [  45, 78923, 3; ...
          8923,     9, 8]
fileName = 'example.txt'
```

After the function runs, a file named `example.txt` should be created containing the following:

```
+----+-----+-+
|45  |78923|3|
+----+-----+-+
|8923|9    |8|
+----+-----+-+
```

Notes:

- The file should not have an extra new line at the end.
- You can run the solution function to see the files created if you need more examples.
- When running the solution function, use a different filename than the one used in your code.

Function Name: `asciiBurger`

Inputs:

1. (*char*) A string containing the customer name and their order

Outputs:

none

File Outputs:

1. A new text file (`'<customer>_order.txt'`) containing a visual representation of the order followed by the price

Function Description:

Following in the latest trends in restaurant chains with highly customizable menus, you decide to open up your own restaurant, `asciiBurger`! With so many options for customers to choose from, the ordering process can become very hectic. You decide to use your MATLAB skills to print an ASCII art representation of each customer's order.

You will be given a text file called '`menu.txt`' with a menu of ascii art for each ingredient and a text file called '`prices.txt`' with their prices. Based on the input, the customer's order specified as a string, it is your task to output a new text file that contains an ASCII image of the customer's order followed by the price of the order printed underneath the ASCII image.

The menu file contains all of the possible ingredients a customer could order. ASCII art for each ingredient will be preceded by a line with an underscore character (`'_'`) and the ingredient name, like this:

```
'_<ingredient name>'
```

Here is an example of a `'menu.txt'` file:

```
_bun
[                  ]
_tomato
oooooooooooooooooooo
_pickles
~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~
_veggie burger
====================
```

The price file will contain one ingredient per line, following the format:

```
<ingredient name>: $<dollars.cents>
```

Here is an example of a `'prices.txt'` file:

```
bun: $0.00
tomato: $0.20
pickles: $500.00
veggieBurger: $2.00
```

Orders will always be in the following format:

`'<customer>,<ingredient1>,<ingredient2>...'`

Based on the given menu and prices, a valid order would be any combination of ingredients, but not necessarily including all of them. If a customer's order is

`'Joe,bun,tomato,veggieBurger'`,

then the cost would be $2.20 and the output text file called `'Joe_order.txt'` would contain the ASCII burger, and the price on the last line:

```
[                   ]
oooooooooooooooooooo
====================
[                   ]
Price: $2.20
```

Notes:

- Assume that every customer who orders a `'bun'` wants a bun on both the top and the bottom of their burger. However, they should only be charged the price of one `'bun'`, as specified by the price in the price list. If a customer does want a bun, it will always be the first ingredient in their order.
- There will not be any invalid orders.
- The ASCII image for an ingredient can exist on multiple, consecutive lines.
- Every line in `menu.txt` has a new line character, including the last line.
- You should print the ingredient art in the order that the customer specifies.
- The name of the output text file should be `'<customer name>_order.txt'`.
- Underscores will never be used in the ingredients' ASCII art.
- The `'menu.txt'` and `'price.txt'` files used to grade are NOT necessarily the same as the ones given to you, so do not hardcode the information from the text files.

Hints:

- The `strtok()` function will be very useful.
- Utilize helper functions, if you so choose.
- Be mindful of newline characters, and when to open and close files.
- You can use `%0.2f` instead of `%d` to display numbers rounded to two decimal places in `sprintf()`.

Extra Credit

Function Name: `nationalTreasure_hard`

Inputs:

1. (*double*) The file handle to the file containing which letters to extract
2. (*double*) The file handle to the file to pull letters from

Outputs:

1. (*char*) A string containing the deciphered code

Banned Functions:

```
fopen(), dlmread(), textscan(), fscanf(), sortrows(), fseek(),
frewind(), fileread()
```

Function Description:

Back in 2004, the possibly greatest actor of our time, Nicolas Cage, starred in possibly the greatest movie of our time, *National Treasure*...wait...this is déjà vu all over again...

After tackling this problem once, we are going to make you think a little harder to make your solution better. You probably opened and reopened files again and again, but why do that if you don't need to?

This problem is exactly the same as it was before except for one thing--instead of the inputs being the names of the files, they are the file handles (what is produced by `fopen()`). What this means is that you can only read through the two text files once because you cannot open and reopen them to read them from the top. What this also means is that you will have to order the information in a cipher in a way such that you can do it all in order in one read through (see hints). The output of the function is a string containing the decoded message from the cipher.

Notes:

- The letter "coordinates" are guaranteed to be valid (i.e. we won't ask for the 5th letter of a word that has only 4).
- Even though you aren't opening the files in your function, **close both files**. If you don't close the files at the end of your function, you won't be able to rerun it as you are debugging it and figuring it out
- Go ahead. Just write the function header then write `fclose()` twice. You won't regret it.
- An easy way to run this function is to call it with `fopen()` in the inputs (see hw08.m file)

Hints:

- The hardest part of the problem is to sort the cipher numbers in order--for this to be done in one run, not only do the lines have to be in order but also the words corresponding to that line have to be in order and the letters corresponding to that word have to be in order.

- Use the indices produced by the `sort()` function and pick which indices that you want to further sort (sort same lines by word and same words by letter).
- The order of the cipher file is the order the final message will be in, but that is not the order you will read the letters from the main text file. Think about how to 'unsort' a vector (i.e. if you do `[sorted, ndx] = sort(vec)` how do you convert sorted back to vec.
- You will probably want to represent spaces in the cipher file in a way to not affect the sorting process.
- Use the smaller first test case as a way to debug and figure this problem out.
- Look at the functions `unique()` and `ismember()`.